Advanced ATARI® BASIC Tutorial

Robert A. Peck

DOS

DOS

advanced BASIC

Advanced
ATARI® BASIC
Tutorial

Bob Peck is a computer consultant, specializing in hardware design, software design and documentation. He has written manuals and diagnostic programs for ATARI and National Semiconductor, among others, as well as articles for various computer publications. He teaches microprocessor assembly language programming. His hobby is the study of computer languages. Among those he uses are BASIC, FORTH, Assembler (for many different processors), Pascal, and C. He received his BSEE from Marquette University and an MBA from Northwestern University.

Advanced
ATARI® BASIC
Tutorial

by

Robert A. Peck

This book is dedicated to my wonderful wife Andrea, without whose patience and understanding this work would not have been possible.

Preface

This book is called the *Advanced ATARI** BASIC Tutorial*. It is designed as a follow-up to the first book in this series, the *ATARI* BASIC Tutorial*. In that first book, you were introduced to many, but not all, of the ATARI BASIC commands. This book will cover the remainder of those commands, and introduce various techniques for either speeding up programs, making them more useful, or, in some cases, accessing certain special features in the ATARI system. This includes making use of occasional machine language patches to your programs for certain types of graphics and other special functions.

Throughout the text, you will find that the techniques used in the first book have been maintained wherever possible. These include the use of *short* demonstration programs wherever possible to minimize your typing, and thorough program documentation to promote your understanding of the basic concepts covered.

It is hoped that you will find the programs both educational and potentially useful in your own program development process.

ROBERT A. PECK

---

*ATARI® is a registered trademark of ATARI®, Inc., a Warner Communications Company.

Contents

Multiple-Way Programming

In *ATARI® BASIC Tutorial,* there was a great deal of concentration on the concept of *menu selection* for programming. This was done as an attempt to show you how programs could be structured to be as friendly as possible to the person who might be using them.

Menu selection, in that section of the book, was controlled by a series of IF statements. Each of the statements compared the menu selection letter to the possible selections, finally causing a jump to the appropriate routine. This section introduces a faster and more concise way of performing the same function. The substitute function is the ATARI BASIC ON-GOTO statement.

Let's look at the technique that was used in the previous book. Assume there is a menu, with selection items A, B, C, and D. Here is one way of deciding which program section would be performed if any of the menu selections are chosen:

```
10 DIM A$(10)
20 REM INSERT MENU TO BE PRINTED HERE
100 PRINT "INPUT YOUR SELECTION (A, B,
     C, OR D)"
110 PRINT "THEN PRESS RETURN."
```

9

```
120 INPUT A$
150 IF A$(1,1)= "A" THEN 400
160 IF A$(1,1)= "B" THEN 500
170 IF A$(1,1)= "C" THEN 600
180 IF A$(1,1)= "D" THEN 700
190 GOTO 100:REM NO MATCH
```

In this case, each of the routines for the processing of the menu selections is assumed to have started at the line numbers indicated in the program piece just shown.

Rather than go directly into the discussion of the ON-GOTO statement, let's first look at the preceding program piece. Since the purpose of this book is not only to tell you about other ATARI BASIC statements, but also to simplify the ways in which you can write your programs, we will take this opportunity to show you simpler forms of certain programs. You may, while you write your own programs, recognize some of these simplifications and use them. Here is a different form of the previous program:

```
10  DIM A$(1)
20  REM INSERT MENU TO BE PRINTED HERE
100 PRINT "INPUT YOUR SELECTION (A, B,
    C, OR D)"
110 PRINT "THEN PRESS RETURN."
120 INPUT A$
150 IF A$= "A" THEN 400
160 IF A$= "B" THEN 500
170 IF A$= "C" THEN 600
180 IF A$= "D" THEN 700
190 GOTO 100:REM NO MATCH
```

This is simplification 1. Notice what has been changed—line 10, and lines 150–180. What has happened here is that the original menu program only cared about the first letter anyway. That is, A$(1,1), starting at character number 1 and extending for 1 character. By changing the DIM statement to A$(1), the program tells ATARI BASIC that regardless of the length of the string the user enters, ignore the rest of the string and keep only the first letter.

Since the length of A$ (pronounced "A-string") is now always limited to one character, the IF statements can be simplified to the form shown. You will be comparing one-character strings to a one-character string.

But even with this kind of construction, you are still limited. The user must always press **RETURN** before you can regain control of the machine. In this manner, the user could enter cursor-move characters, and many other things which would make it difficult to properly interpret his input, or which would cause the user to mess up your display. This subject was discussed in the previous book.

Here is a far better solution to the problem. It uses two new concepts: (1) the OPEN statement and (2) the GET statement. Each of these statements will be used further, later in the book, especially in the section devoted to *disk operations*. However, both statements come in very handy here in showing you how to keep control of the program process.

Here is another version of the previous program which uses the new statements. It also uses the ON-GOTO statement in the final simplification. The explanation of each of the statements used here follows the demonstration program segment. A more complete example is shown in this case because less overall typing is required.

```
10  OPEN#1,4,0,"K:"
20  REM INSERT MENU TO BE PRINTED HERE
99  POSITION 3,20
100 PRINT "YOUR SELECTION (A, B, C, OR
    D)?";
110 GET#1,X
120 Y=X-ASC("A")+1
130 ON Y GOTO 400,500,600,700
140 GOTO 99
400 Q=400:GOTO 1000
500 Q=500:GOTO 1000
600 Q=600:GOTO 1000
700 Q=700
1000 PRINT "PROGRAM WENT TO LINE ";Q
```

The command in line 10, OPEN#1,4,0,"K:", said to reserve an Input/Output Control Block (called IOCB) for our use. There are eight possible IOCBs in ATARI BASIC. Numbers 0, 6, and 7 are normally in use by the system. Numbers 1–5 can be used for other purposes such as this one.

An IOCB is a special area in memory which can be reserved by this kind of statement to aid in performing certain kinds of system-controlled commands. These include getting characters, getting input "records," writing out characters and records, and certain special operations called *device-dependent* commands. This means commands that only one kind of device can understand and perform. Such examples are *rename* a *file* or *erase* a *file*. These and others are covered in this book in the section on *disk*-access *operations*.

The second part of the command, OPEN#1,4,0,"K:", the 4, is used to tell ATARI BASIC that this block is to be used for *input only*. Other things the block can be used for are: *output* (as in OPEN#1,8,0,"K:") or *input and output* (as in OPEN#1,12,0,"K:"). Since this is the keyboard, only input is allowed. Therefore, the number 4 command has been placed in that position.

You can tell that this block is being reserved for the use of the keyboard by the last part of the command, the "K:", which defines the ATARI keyboard.

The third part of the command, the 0, is in the position for what is called the *device-dependent code*. For most of the ATARI devices, this part of the command will *always* be zero, so we won't worry about this part until a command is used requiring a nonzero value.

The purpose of the command is to provide an alternative means of getting data from the keyboard. Normally the keyboard is firmly linked to the ATARI Screen Editor. The Editor, as you know from experience using the INPUT statement, requires that the user touch **RETURN** before its data is accepted. Once you establish the keyboard as a separate entity, though, you can GET keystrokes, one at a time, without pressing **RETURN** for any of them.

The other advantage to this method is that the keystrokes do not, as a result of the GET statement, appear on the screen. You

can translate them any way you wish, and specifically control what is written to the screen as a result of a user-entered key.

The command in line 110, GET#1,X, means that now that the Input/Output Control Block (IOCB) is OPEN, it can be used as a pointer to the device. The GET statement tells the computer to GET the next available input data byte and assign its value to the variable called X. Then GET#1 means that IOCB number 1 is to be used as the pointer to the device from which the data is to be retrieved.

All ATARI-compatible devices are linked to the computer's operating system by a set of common command elements. Therefore, once an IOCB has been tied to a specific device, there is no need for the software to know any of the characteristics of the device itself. All that needs to be known is that the GET statement will retrieve, from the device, the next available data byte.

When the keyboard is read, the ATASCII key value will be placed in the variable called X. Now this value can be treated just as any other value. You may, if you wish, try the following short program which demonstrates only the result of the GET statement:

```
10 OPEN#1,4,0,"K:"
99 POSITION 3,20
100 GET#1,X
200 PRINT " ";CHR$(X)
1300 GOTO 99
```

This program has the effect of placing the cursor at a fixed position, then printing the character equivalent of the key touched in the position to the immediate right of the cursor.

This is the reverse of normal where the cursor always trails the character printed. What happens here is that the keyboard input is never shown on the screen except by directly using the PRINT statement. Therefore, if you wish, you could PRINT anything in response to the various kinds of keys you can show as possible inputs.

The next statement in the program is

```
120 Y=X-ASC("A")+1
```

What happens here is that the character E is converted to its ATASCII value, which is 69. The capital letters of the alphabet, in ATASCII values, start at 65 and go for a total of 26 values in ascending order. Therefore, for any letter to be converted to its numerical position, 1–26 in the alphabet, it is necessary to sub-tract the ATASCII value of the letter A from the letter value re-ceived, then add 1 to it. This is what this program does.

If the possible selections are represented by the letters A–D, then it gives us four possible "index numbers" as a result of these four inputs. These are the numbers 1, 2, 3, and 4. They are put to use in the following statement:

```
130 ON Y GOTO 400,500,600,700
```

When ATARI BASIC sees this kind of statement, it looks at the value of Y and knows it is supposed to GOTO the line number at the position number which matches Y. In other words, if Y is 1, GOTO the line number that is in the first position. If Y is 2, GOTO the line number in the second position, and so on. There are, in this sample statement, four possible positions, so the values of Y from 1 through 4, included, have a meaning. This is a more compact way of expressing the IF-THEN construction shown at the beginning of this section. Since it is more compact, it can also be faster to operate.

Any value of Y that is not in the range from 1 to 4 will be ignored by the program since there is no line number for those positions. This allows you to maintain control because only single key-strokes can be accepted at a time. If they are not acceptable (not recognized as meaningful), the user is again prompted to give one of the correct replies.

The last part of the sample program, lines 400–1000, simply shows you that the sample transfer of control to the new line number happened "as advertised."

ATARI BASIC also offers the same kind of construction for sub-routine calls. In that case, in place of the statement

```
130 ON Y GOTO 400,500,600,700
```

you might have the statement

```
130 ON Y GOSUB 400,500,600,700
```

where each of the routines to which the statement points is terminated by a RETURN statement.

Note that in the ON-GOTO statement, the program control actually *fully transfers* control to the new line number. In the case of the ON-GOSUB statement, control is only transferred *temporarily* to the routine starting at the selected line number. On execution of the RETURN statement, control will return to the next line following the line which called the subroutine. (NOTE: All subroutines used with an ON-GOSUB statement must end with a RETURN statement.)

As an exercise, you might consider what kind of program it would take to simulate a simple calculator, using the GET statement and the ON-GOSUB statement. (Hint: So-called reverse-Polish calculators are probably easier to simulate than the others.) An example program flow description follows:

1. Set variables A and B to 0 (use A as the running TOTAL, B as the current input).
2. OPEN the keyboard.
3. GET a keystroke.
4. Is it a number (ATASCII "0" through "9")? If yes, multiply current value of B by 10 and add the value of the input (limited to integer values only for this not-so-complex version), then go back to Step 3.
5. Is it a plus sign (ATASCII " + ")? If yes, add the current value of B to the current value of A and display the running total = A, then go back to Step 3.
6. Is it a minus sign (ATASCII " − ")? If yes, subtract the current value of B from the current value of A and store and display the result as the running total = A, then go back to Step 3.
7. Is it a times sign (ATASCII "*")? If yes, multiply the number in B by the number in A, store in A, and display the result as the running total = A, then go back to Step 3.
8. Is it a divide sign (ATASCII "/")? If yes, then divide the number in A by the number entered in B, store the result in A, and display it as the running total = A, then go back to Step 3.
9. Is it a capital C (ATASCII "C")? If yes, this means "CLEAR."

Set both A and B to zero, display the value of A as the running total = A, then go back to Step 3.

This little narrative example can be expanded, of course, to include handling of decimal input numbers and other kinds of calculations, such as squares, square-roots, and so forth. The example is just placed here to tickle your imagination a bit.

Remember also that you should include some kind of error trapping in a program wherever necessary. This lets you retain some measure of control, even if the user finds a way to make a mistake.

ERROR HANDLING IN MULTI-WAY BRANCHES

In the book, *ATARI BASIC Tutorial,* you would have encountered a recommendation that programs be written as though they are a set of functions. If each of the functions can be individually tested in some way, then when they are put together, the resultant program should be more easy to debug. (If all of the program pieces work OK, then if a single program is made by a controlling program that calls each piece, most everything should work together, providing there are no direct conflicts between the program pieces in the first place.)

This technique is sometimes called *structured programming.* In other programming languages, a more truly structured programming can be attained, but in ATARI BASIC, this organization as a set of subroutines comes close to a "structure," so we will reference it loosely this way.

When you develop structures like this, you will wind up with a potential problem in controlling your error handling. Let's look at an example:

```
10 REM CONTROLLING OVERALL PROGRAM
20 GOSUB 2000
30 GOSUB 4000
40 GOSUB 6000
50 GOTO 20

2000 REM PRETESTED ROUTINE THAT FORMS A
       USER MENU
```

```
2999 RETURN
4000 REM PRETESTED ROUTINE THAT
     PROCESSES DATA
4001 REM ENTERED BY THE MENU PART
4999 RETURN
6000 REM PRETESTED ROUTINE THAT WRITES
     A REPORT
6001 REM BASED ON THE PROCESSED DATA
     FROM 4000,
6999 RETURN
```

Now, if you are trying to maintain control of the overall process, you will want to TRAP errors that can occur. Where should you put the TRAP statements? And where should you place the error-handling routines?

For various kinds of data entry routines, when you want to give the user another chance to get it right, you will normally want to put the TRAP statement and its error handling "locally" to the subroutine that is actually being used to retrieve the data. In other words, you might have the TRAP set before a user input in subroutine 4000, for example, while the error handling is somewhere within the routine itself.

Likewise, for the printing of a report, a pretested subroutine 6000 would probably have a TRAP statement preceding the PRINT statement. For example, if a program discovers an error during printer communications, such as the printer is off, you could handle user communications with a routine that asks, "Do you really want to print this now? If so turn on the printer and press **RETURN**. Otherwise, press S to save the report for later, or Q to quit."

There are, however, cases where the error happens somewhere that cannot be handled so easily. For example, if you open a disk door while a file is being written, or if there is some other kind of error somewhere in the middle of the program process, the logical thing to do is to TRAP these errors into the main lines of the program itself (in the example, lines 10–50 somewhere).

What the error reporting will say is, "Error in process number . . ." or some other kind of message. It can then offer whatever best alternate route is available, based on the error that occurred.

Maybe it would not be necessary for the user to re-enter all of the data. Maybe the disk was full on the processing program and a new disk with more room would solve the problem. Then the program could proceed with the process again, calling which-ever subroutine (using a GOSUB and RETURN) would be re-quired to get things back on track.

MAIN PROGRAM (called level 1)

statement

statement

GOSUB 4000 ◄───────────── Passes control to line 4000, but saves "on a stack," the location of whatever statement follows this GOSUB so that a RETURN statement, when sensed, will RETURN to this next location.

This case is a level-2 subroutine.

next-statement-A

GOSUB 6000 ◄───────────── Performs same action as for routine 4000. If RETURN happened OK for routine 4000, then only the RETURN ad-dress for routine 6000 is on the stack.

This case is also a level-2 subroutine. If this routine called another, it would be at level 3, and so forth.

next-statement-B

level-1 TRAP error handling

level-2 TRAP error handling

(These are the things we are interested in at this time — how to handle those TRAPs that return control to level 1 without issuing the RETURN.)

Fig. 1-1. Possible problems with TRAPs in subroutines.

However, when you TRAP to a statement outside the range of the subroutine itself, and do not execute a RETURN to get back to the "caller," there is a possible problem. This is illustrated by the diagram in Fig. 1-1.

This is what happens to the system if TRAPs occur without RETURN statements being recognized. Fig. 1-2 is a sample of a stack appearance when the TRAP sends an error condition up to a level-1 routine from a level-3 routine. (Example: If the menu routine does another GOSUB, and an error occurs in that final level of GOSUB and is trapped to a common error handler in the main routine.)

Each time you perform another level of GOSUB without first issuing a RETURN, another so-called RETURN address gets added to the stack, and the stack pointer moves down one notch to point to the next place it can use to save the next level of RETURN address.

If you handle the errors, then perform more and more GOSUBs without RETURNs, then eventually the stack runs out of room to store these RETURN positions and the program will fail. Notice that each time you issue a RETURN, the pointer moves up again and reuses a previous position. Therefore, if the GOSUBs and the RETURNs are matched, you can perform an indefinite number of each without causing a problem.

How, then, can you handle errors at level 1, when necessary, and yet prevent this problem from causing program failure? Well, this is where the ATARI BASIC statement called POP is used.

The purpose of POP is to move the pointer up one position each time the POP is performed. This means that if all of the error

SYSTEM STACK ——————▶ Where the RETURN locations are kept.

START POINT ——————▶ RETURN address from level 2 to level 1.
                                              RETURN address from level 3 to level 2.

STACK POINTER ——————▶ Next special memory area where another
                                              level of GOSUB RETURN address can be
                                              stored.
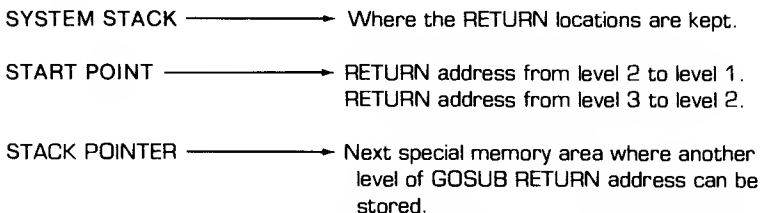
Fig. 1-2. Stack appearance when TRAP sends error condition from level-3 to level-1 routine.

handling is TRAPped up into some part of the level-1 routine, then you would perform as many POP statements as the subroutine was "deep" in levels at the time the error happened. An example is shown here:

```
10 REM MAIN PROGRAM
20 GOSUB 4000
......    (level 1)
50 POP:POP:POP
......
......          .
4000 ......   (at level 2)
     GOSUB 8000
......
8000 ......       (at level 3)
     GOSUB 10000
......
10000 ......          (at level 4)
      TRAP 50
```

(Here somewhere is the condition causing the actual error.)

Line 50 brings the pointer from level 4 to level 3 with the first POP, level 3 to level 2 with the second POP, and level 2 to level 1 with the last POP. Level 1 is the level at which there are no RE-TURN addresses in the stack, so the stack is now cleared for additional use, and the errors can be reported and handled as you wish.

The rule is that if there is no RETURN performed from a subroutine, then a POP should be done instead. It keeps the program healthy and avoids unnecessary and difficult-to-find crashes.

REVIEW OF CHAPTER 1

1. It is possible to make programs shorter and perhaps faster if one uses the most efficient way of performing a task. This includes an awareness of what the different kinds of ATARI BASIC statements will do and how they can be used.
2. The ON-GOTO and ON-GOSUB statements can be used for

multi-way branching. Each will only perform its task if the control variable exactly matches one of the numerical positions where a destination line number has been shown.

3. The GET statement can be used to accept data from the user without pressing **RETURN** , offering a greater degree of control over what will appear on the screen.

4. One must be careful when handling errors at various levels of subroutines. Otherwise, the system may crash mysteriously. The POP statement is used for this purpose.

CHAPTER 2

Advanced String Handling

In the *ATARI BASIC Tutorial,* a basic explanation of strings and string handling was given. This chapter will cover the topic from a somewhat different angle. In particular, this chapter will try to emphasize how ATARI BASIC compares to other versions of BASIC. If you have developed an interest in programming, it is likely that you have seen programs in magazines written for and about other variations of BASIC. This chapter will attempt to show you how ATARI BASIC works in comparison to the others.

HOW TO RESERVE SPACE FOR STRINGS

In ATARI BASIC, it is necessary to save space for strings. Other BASICs do not need this, and allow dynamic allocation of string space. There are two advantages to the ATARI BASIC approach to strings, one is that the string handling will not result in "garbage collection" delays. (You may find various magazine articles which deal with this problem.) ATARI BASIC, because of saving space for strings directly, does not encounter this delay.

The other advantage is that once the program begins running, the location of the string start in memory becomes fixed. Therefore, you may, if you wish, use the string to contain machine

language instructions. Examples of this will be found later in this book.

Here is an example of specifying space for a string:

```
10 DIM A$(100)
```

This line saves a space of a maximum of 100 memory locations for the string named A$. No matter how many characters are stored there (from 0 to 100), ATARI BASIC will always keep that same space reserved for that string.

TRUNCATING A STRING

The term *truncating* means cutting off something at something less than its maximum length. ATARI BASIC, on accepting data for a string, will only accept the maximum number of characters that are reserved in the string space. The rest of the characters entered are lost. An example is shown here:

```
10 DIM A$(6)
20 PRINT "ENTER 6 OR MORE CHARACTERS"
30 PRINT "ONLY THE FIRST 6 WILL BE
   ACCEPTED"
40 INPUT A$
50 PRINT "I ONLY STORED: ";A$
```

HOW THE LENGTH OF A STRING IS DETERMINED

You may, at any time, find out the current length of a string by executing the LEN command, as shown in the following example:

```
60 PRINT "THE LENGTH WAS: ";LEN(A$)
```

If, when a string item is requested, only the **RETURN** key is pressed, the length of the string is called zero. Each character entered advances the count by 1. If the backspace key or an editing key is pressed, the count is changed according to what you have done to the screen. The minimum length will be 0, which can also be caused by initializing a string by a statement, such as:

```
70 A$ = "":REM TWO SETS OF QUOTES,NO
   SPACE BETWEEN
80 PRINT "NOW THE STRING LENGTH IS:
   ";LEN(A$)
```

In some of the chapters that follow, you will see examples where strings are formed individually by characters separately entered. The main reason for this is to maintain control of the data entry process (accept good characters, reject bad characters). However, this formation can only build a string to the maximum length provided for the string. Otherwise an error is generated.

HOW TO REFERENCE AN ENTIRE STRING

To reference the entire contents of a string in ATARI BASIC, you simply use the name of the string. For example,

```
PRINT A$
```

prints the whole thing.

HOW TO REFERENCE THE RIGHTMOST CHARACTERS

In other versions of BASIC, there is a function called RIGHT$. ATARI BASIC does not use this function; it can reference the rightmost characters simply by specifying in parentheses the starting character of the group to be referenced. The total number of characters used will start at this point and proceed to the maximum number of characters currently in the string. An example is:

```
10 DIM A$(6)
20 A$ = "ABCDEF"
30 PRINT "THE RIGHTMOST 3 CHARACTERS
   ARE: ";A$(4)
```

which prints DEF as the result when the program is RUN.

This is actually a short form of the string specification shown in the next section. In other words, the starting character is specified, and the ending character is "understood."

HOW TO REFERENCE THE LEFTMOST CHARACTERS

In other BASICs, there is often a function called LEFT$ which separates from the string a specific number of characters starting from the first one. ATARI BASIC performs this function by accepting, in parentheses, two parameters. These are the starting and ending character for the string split. Note that this also allows the selection of numbers of characters out of the middle of the string as well. Here is an example:

```
10 DIM A$(6)
20 A$ = "ABCDEF"
30 PRINT "LEFT 3 CHARACTERS ARE:
   ";A$(1,3)
40 PRINT "CENTER 2 CHARACTERS ARE:
   ";A$(3,4)
50 PRINT "RIGHT 3 CHARACTERS ARE:
   ";A$(LEN(A$)-2, LEN(A$))
```

The last line (line 50) illustrates that the values you give to the string position function can either be numbers or functions. Line 50 says that whatever the length of the string, this line is to print the last three characters (last − 2, last − 1, and last). The other way to write this line is to have it print A$(LEN(A$) − 2), since the second variable will be understood.

HOW TO ADD STRINGS TOGETHER

This is the function of stringing things one onto another. ATARI BASIC does not use the STRING + STRING kind of function. Instead, you must specify where the strings are to be added together. They can be added, one onto the end of another, by the following program sample:

```
10 DIM A$(100), B$(6)
20 A$ = "ABCDEF"
30 B$ = "1234"
40 A$(LEN(A$)+1) = B$
```

where the length of A$ is 6. Therefore, B$ will be added onto and stored in A$ starting at position 7 and going for the exact current

length of B$. This means that even though there are six positions reserved for B$, only four are occupied, which makes the current length of B$ equal to 4. Therefore, the result in A$ will have a length of 10, and be composed of the characters ABCDEF1234. It is not necessary to add onto the end of the string; you can specify any starting position you wish, such as:

```
10 DIM A$(100),B$(6)
20 A$ = "ABCDEF"
30 B$ = "1234"
40 A$(3)=B$
```

This example simply replaces part of the previous part of A$ with the new contents of B$. The result in this case, if A$ is printed, will be AB1234. You may use this function in forming words or sentences in some future program. (See "Adding a Character to the Middle of a String.")

MAKING A STRING SHORTER

This is a little trickier than adding to the string. There are two possible approaches to shortening a string. One is more obvious, the other takes advantage of one of the features of ATARI BASIC to save some program space. Here they are:

Shortening a String (Version 1)

```
10 DIM A$(100)
20 DIM B$(6):REM MAX LENGTH IS 6 TO USE
30 A$ = "SHORT TEST OF STRING
   SHORTENER"
40 B$ = A$
50 A$ = B$
```

In this first version, A$ is long. When B$ = A$ is executed in line 40, only six characters can be held in B$, so its maximum length will be 6. When line 50 is executed, each character of A$ is made equal to each character of B$, with the final length of A$ being equal to the length of B$. Thus, A$ has been made shorter.

Shortening a String (Version 2)

```
10 DIM A$(100)
20 DIM B$(6):REM MAX LENGTH IS 6 TO USE
30 A$ = "SHORT TEST OF STRING
   SHORTENER"
40 PRINT "HOW MANY CHARACTERS FOR
   LENGTH"
50 INPUT N
60 A$(N) = A$(N,N)
70 PRINT A$
```

On the right side of the equals sign in line 60, it says A$(N,N). In previous examples, the use of two variables in the parentheses specified the starting and ending characters for a string piece selected in this way. This form then specifies that a string piece starts and ends on the same character. Therefore, this function specifies a single-character string.

On the left side of the equals sign in line 60, it specifies that the starting point for adding the string piece is to be character number N. Therefore, the character at position N alone is placed at position N in the string. Since this is the last position that is to be modified by ATARI BASIC, it assumes that this is the length of the string. Therefore, the string has been shortened.

DELETING FROM A STRING

Taking a character out of a string is as easy as determining the position of that character, then telling ATARI BASIC to write over it, as:

```
10 DIM A$(20)
20 A$ = "123456789" : PRINT A$
30 PRINT "WHICH ONE TO DELETE"
40 INPUT N
50 A$(N) = A$(N+1)
60 PRINT A$
```

In this example, on the right side of the equals sign in line 50, it is understood that the string specification A$(N+1) actually

means A$(N + 1, LEN(A$)). This means that all characters to the right of the character at position N will be moved one position to the left, covering up location N, and making the string one position shorter.

ADDING A CHARACTER TO THE MIDDLE OF A STRING

This is the reverse of the preceding section. It will move characters one position to the right. The main difference is that it requires another string, as long as the one being used, to accomplish the task. In other words, it needs a place to store the characters from the one selected to the end of the string so they can be put back again in the right place.

```
10 DIM A$(20),B$(1),TEMP$(20)
20 A$ = "123456789":PRINT A$
30 PRINT "ADD AT WHICH POSITION"
40 INPUT N
50 B$ = "X"
60 TEMP$ = A$(N)
65 REM SAVES THE RIGHTMOST CHARACTERS
70 A$(N) = B$
75 REM ADDS THE NEW CHARACTER
80 A$(N+1) = TEMP$
```

REVERSING THE ORDER OF CHARACTERS IN A STRING

This topic will be shown from two different approaches. The first one will show the handling of the string data as a string itself. The second approach will introduce a new function of ATARI BASIC. First, let's set up the string to be used:

```
10 DIM A$(10)
20 A$ = "ABCDEFG"
```

Now create some space for other string items that will be needed for the reversing operation. (The goal is to take ABCDEFG and produce GFEDCBA.)

```
30 DIM B$(1),C$(10)
```

In line 30, B$, a single-character string, will be used to grab one character at a time and place it into the output string called C$.

```
40  LA = LEN(A$):J = LA
50  FOR I = 1 TO LEN(A$)
60  B$ = A$(J)
```

Lines 40 through 60 tell the computer to accept a single character starting at position J, and throw the rest of the string away.

```
70  C$(I) = B$
```

Line 70 says to write one character, starting at position I. (Characters to the left of position I are not touched in C$. All it does is add one to the string length each time the process is repeated.)

```
80  J = J - 1
```

Line 80 sets the pointer to the next earlier character.

```
90  NEXT I
```

Line 90 says to repeat line 80 until all characters are moved into the new string.

```
100  PRINT C$
```

Finally, line 100 tells the computer to print the result.

That was the first approach. It required an additional string, at least as long as the first one. It also required I passes through the program loop, where I is the total number of characters that were in the string. You may, at times, wish to move characters *within* a single string, not using an external string as large as the first one. Here is one way you can do that:

```
10  DIM A$(10)
15  A$ = "ABCDEFG"
20  DIM B$(1),C$(1)
```

Line 20 uses single-character strings for this part.

```
30  K = INT(LEN(A$)/2)
```

Line 30 will provide an ending point that is only half the number of cycles to work than the first approach. Since the goal is the same, to change ABCDEFG into GFEDCBA, this program will, instead of moving all seven characters into a new string, swap them around within the old string. Two character moves will be done during each loop, so only half the number of loops will be required.

```
40 RTPOS = LEN(A$)
```

Line 40 defines which character position is the rightmost in the string that is to be shifted around.

```
50 FOR I = 1 TO K
60 B$ = A$(I,I)
```

Lines 50 and 60 tell the computer to grab one character from the left side of the string. (The loop constant specifies which one to use.)

```
70 A$(I,I) = A$(RTPOS+1-I)
```

Line 70 says to move a right-hand character into the left-hand position.

```
80 A$(RTPOS+1-I) = B$
```

Line 80 says to move the left-hand character into the right-hand position.

```
90 IF I = 1 THEN C$ = B$
```

Line 90 will be explained later.

```
100 NEXT I
```

Line 100 says do it until done.

```
110 A$(RTPOS) = C$
```

Line 110 declares that the string is as long as it was before.

```
120 PRINT A$
```

Line 120 tells the computer to print GFEDCBA.
You will notice that the order of swapping of the string pieces

was: (1) grab a left character (line 60), (2) move a right character (line 70), and (3) put the left character where the right one was (line 80). This is the *required* order because ATARI BASIC sets the length of a string by means of which character was written *last*. In other words, if you tried to grab a right character, move a left character to the right character spot, then put the right character back where the left character was located, the loop will only work one time. This is because after only one pass through the loop, the last known length of the source string is one character, since it is the last one written. The data will not be disturbed within the string storage space, but ATARI BASIC will believe that the length of the string stored there is only one character.

The drawback to using this approach is that each time the loop is executed, the string gets one character smaller. It starts out by swapping the outermost characters, then the next innermost, and the next until the center of the string is reached. Once this happens, the string looks as though it is only half as long as it started out. (Delete line 110 and RUN the program again to see what is meant here.)

The reason for line 110 is to tell ATARI BASIC that the string should actually be the same length as the original. ATARI BASIC, as mentioned earlier in this chapter, will add characters to a string starting at a specific location and proceeding for the length of the string element being added. It does not disturb any of the characters previously placed in the string from number 1 through and including the one just prior to the position being added.

Line 90 saves the rightmost character for the swapped string. If I = 1, then B$ contains the original lead-in character from the source string, which becomes the last character in the string. Line 100 will, when I = 1, store that character in the last position. But then for each pass through the loop, it stores a single character in the preceding position. This makes the length of the string, as seen by ATARI BASIC, to be one less than it was before.

Because B$ is only one character long, only one position is filled and the other characters are undisturbed. This means that the swapped string will be properly formed, but the effective length will be wrong. Line 110, then, will not be changing the value placed at the last position in this string because it will

replace an A with an A (found and placed on the first pass through the loop). However, because the A will be written into the last position in the string, it will tell ATARI BASIC that this is the length of the string and complete the task.

Both of the preceding techniques may be thought of as "first approach" techniques because they use ATARI BASIC string handling directly and, therefore, must either loop more times than may be necessary, or use more memory space, or need to fool ATARI BASIC into thinking that some different string length is being used.

The sample program that follows in the next section can swap a string where it sits, without changing what the system sees as the length of the string. It may also be used to selectively change any one of the string variables without fooling around with re-adjusting the length.

You can even use string storage spaces as direct storage places if you wish. Note that you must be more careful in using this technique since the statements used here can put data any-where in memory. This means that your program could "crash" the system if it is not properly limited in its memory POKEs.

INTRODUCING THE ADR FUNCTION

When you use the DIM statement to assign a storage space for a string, the system, on starting a RUN, will assign a space for that string to be stored. During the RUN of the program, the place reserved for that string will not be moved by ATARI BASIC. You can find the location of the string by the following example state-ment sequence:

```
10 DIM A$(20)
20 A$ = "ABCDE54321"
30 X = ADR(A$)
35 PRINT X
```

Lines 30 and 35 will show you where in memory A$ has been placed. (If you are only interested in where in memory a specific character has been placed, use the function as: X = ADR(A$(N))

where N will be that character position number in which you are
interested.)

```
40 FOR I = 1 TO LEN(A$)
50 PRINT "FOUND A$(";I;") IS:
   ";CHR$(PEEK(I+X-1))
60 NEXT I
```

Lines 40 through 60 demonstrate that the character string data
can be located directly by performing a PEEK at the address
where the ADR function says the character string data starts.

Likewise, you can store a new set of character data by doing
a POKE into the address at which you wish to store the new
characters. If, for example, you wish to change characters 2 and
3 into question marks, you could add lines 70–100 to the sample
program and perform that task.

When you perform the last line of this part of the sample pro-
gram, notice that the entire size of the original string is printed.
This happens even though you have written into the string some-
where in the middle. You see, you did not use the string writing
function itself, so ATARI BASIC did not realize you had written a
string, and it did not change what it thought was the string length.
Here are the lines that will change characters 2 and 3:

```
70 QM = ASC("?")
75 REM USE THE ASC FUNCTION TO GET THE
   ATASCII VALUE OF "?"
80 FOR I = 2 TO 3
90 POKE (X+I-1),QM
100 NEXT I
200 PRINT A$
```

X is the actual machine location where the first character of A$
(which is A$(1)) is stored. Therefore, $X + I - 1$ equals that location
plus 2 minus 1, which means one greater than the starting char-
acter, which means character number 2 of the string.

Now this technique can be used to rewrite the string reversing
function:

```
10 DIM A$(10)
15 A$ = "ABCDEFG"
30 K = INT(LEN(A$)/2)
```

As in the last example, line 30 provides the ending index point. As in that example, instead of moving all seven characters into a new string, swap them around within the old string. Two character moves will be done during each loop, so only half of the number of loops will be required.

```
40 RTPOS = LEN(A$)
```

Line 40 defines which character position is the rightmost in the string that is to be shifted around.

```
45 WHERE = ADR(A$)-1
```

Line 45 subtracts 1 from the starting point where the data is stored so it is possible to reference the characters by calling them "number 1," "number 2," and so forth. Adding one to WHERE points to number 1, etc.

```
50 FOR I = 1 TO K
60 TEMP = PEEK(WHERE+I)
```

Lines 50 and 60 are used to grab one character from the left side of the string. (The loop constant specifies which one to use.)

```
70 POKE WHERE+I,(PEEK(WHERE+RTPOS+1-I))
```

Line 70 moves a right-hand character into the left-hand position.

```
80 POKE WHERE+RTPOS+1-I,TEMP
```

Line 80 moves the left character into the right position.

```
100 NEXT I
```

Line 100 says do it until done.

```
200 PRINT A$
```

Line 200 says PRINT the results.

The right-hand character is grabbed by finding its address in the memory and PEEKing the contents. The address is formed by the location WHERE + RTPOS, which points to the rightmost character. Next it adds 1 and subtracts I, which moves left one position each time the loop occurs.

When line 200 executes, you will see the same result as a reversed string, but no actual string-type instructions have been called in ATARI BASIC. Therefore, the length information will not be affected.

REVIEW OF CHAPTER 2

1. The string functions available in ATARI BASIC require that you assign space for strings before they can be used.
2. The string functions allow you to isolate characters from any part of a string; left, right, or middle.
3. The length of a string in ATARI BASIC depends on which character number is the last one written.
4. You can change the data within a string either by using the string functions or by finding out where the string is stored in the memory and using a set of PEEK and POKE statements to change the data. This does not affect the length of the string as far as ATARI BASIC is concerned.

CHAPTER 3

Using the Disk System with ATARI BASIC

In the *ATARI BASIC Tutorial,* the disk system was introduced primarily as a place where your programs could be stored and loaded. In that book, the commands LOAD and SAVE were the only ones formally introduced. This book assumes that you have access to a disk system and concentrates heavily on the use of the disk within the program.

This chapter introduces the concept of *ATARI DOS,* then proceeds to explain each of the ATARI BASIC commands which use DOS. First, DOS is an abbreviation for Disk Operating System. This is a program which is automatically loaded into the computer when it is first turned on. In order to assure that the DOS will be available for your use, always be sure that the following steps are followed when you power up your system:

1. Be sure, if you have more than one disk unit, that one of them is switched as unit 1, and that each of the others has a separate, different number. Make sure the cables are connected as indicated in your ATARI DOS manual.
2. Insert a diskette into drive 1 and close the door. This diskette must contain a file called DOS.SYS if you are using ATARI DOS-2 or, if you are using ATARI DOS-3, the disk contains the files KCP.SYS and FMS.SYS.

3. Turn on the disk unit before you turn on the computer. This is because the ATARI disk unit is a "smart peripheral." This means that it is capable of responding to commands. When the computer is turned on, it sends a command out to all devices connected to it, saying "are you there?" If the device does not respond, the computer assumes that the device is not connected. If a device does respond, the computer will continue to communicate briefly with it to load more information directly from the device. In the case of the disk unit, the computer will try to load ATARI DOS from the disk. This prepares the system for the ability to use the disk commands. This means that if the disk unit is not turned on first, the computer will assume it is not connected at all, and will ignore it. Since you will want to use the disk commands, be sure to follow the correct sequence.

4. Now that each disk unit is on, turn on the computer. Assuming that the ATARI BASIC cartridge is in place, the disk operating system, or DOS, will be loaded and the DOS commands can be used.

HOW TO SAVE PROGRAMS

After you have developed a program, you may want to save it for future use. There are two different ways of saving a program. One of them uses a command called SAVE. The other uses a command called LIST.

With the SAVE command, the program will be written to the disk in a form that only the BASIC cartridge can read. It is a form called *tokenized,* and is kind of a shorthand. Once the program has been brought back into the machine by the BASIC cartridge, you can LIST it on the screen, printer, and so on. The token form of the program takes up less space because each of the ATARI BASIC keywords will have been replaced by a single character. This character may be reinterpreted by the BASIC cartridge as the keyword later, either for you to see, or simply be used directly, during a RUN command, to perform the desired function.

An example program is shown here and may be used for practice:

```
10 PRINT "THIS IS A PRACTICE PROGRAM"
```

To SAVE this program, enter the following command:

```
SAVE "D1:TEST"
```

and press **RETURN**.

A program saved this way can be brought back into the machine using the LOAD command, such as:

```
LOAD "D1:TEST"
```

assuming the test program is still present in the machine after you SAVE it.

Type the following to test the save and load sequence:

```
10 PRINT "PRACTICE PROGRAM"
SAVE "D1:TEST"
NEW
LIST
```

Since you typed NEW, the screen should just show READY, saying that no program is present.

```
LOAD "D1:TEST"
RUN
```

This should print:

```
PRACTICE PROGRAM
```

There is another way in which a program may be saved on the disk. This is done with the LIST command. Normally you would use LIST just to see all or part of your program on the screen, such as

```
LIST 1,100
```

which would list the contents of lines 1 through 100 on the screen. This LIST command may also be used with a printer, for example, to list the program on the printer. In this case, the form of the list command would be:

```
LIST "P:"
```

This lists the entire program to the printer, or

```
LIST "P:",1,100
```

This lists just lines 1–100, where P stands for the printer device.

The ATARI operating system is very versatile because it allows the data output or input to be directed to, or accepted from, many different kinds of devices. All that is needed is that you specify, with the command, enough information about where the data is to go, or come from, and the operating system handles the rest.

For the LIST command, then, the structure will be as follows for LISTing a program onto the disk. Note that in this command, the data that is sent to the disk is exactly what appears normally on the screen. In other words, the program is stored as a set of strings, or sentences, if you prefer. Because it is this kind of data, it may be processed in a special way if desired.

In the remainder of this chapter you will see examples which use, as their data files, programs that have been LISTed to the disk. These examples are in the section covering the OPEN and INPUT statements.

You can LIST a program to the disk using the following command:

```
LIST "D1:LONGTEST"
```

This will list the entire program onto the disk in unit 1. You could also use the LIST command by typing:

```
LIST "D1:LONGTEST",1,100
```

This will list only lines 1 through 100.

Programs which have been LISTed to the disk may be brought back into the machine using the ENTER command. An example is shown here, using the same program name used above. First type NEW, then type:

```
ENTER "D1:LONGTEST"
```

In addition to the uses you will see later, the LIST and ENTER commands have a special property that you may find very useful. The property is that the ENTER command, as far as programming

is concerned, treats the source device in exactly the same way as it would treat the keyboard. In other words, ATARI BASIC stays in the command input mode, and accepts from the source device whatever is "out there," just as though additional keystrokes were being entered from the keyboard. Unlike the LOAD command, which executes a NEW before it does the actual LOAD, the ENTER command *does not disturb any program piece that is already present in the machine!*

You may use this fact to build custom programs out of pieces you have developed and LISTed separately to the disk. For example, you may have developed a special title block routine which you have numbered lines 28000 through 31000. You may also have a special input routine which you have numbered lines 20000 through 24000. If these have been listed separately to the disk by the commands:

```
LIST "D1:PART1",28000,32000
```

and

```
LIST "D1:PART2",20000,24000
```

or by any other separate LIST command, then you can bring *both* parts of the program back into the machine by the set of commands:

```
ENTER "D1:PART1"
```

and

```
ENTER "D1:PART2"
```

NOTE: Since these are treated as though they are entries from the keyboard, if there are any duplicate line numbers in the data, the last-entered version of the line number is the one that will appear in the final version of the program.

In this manner, you can develop separate subroutines for each of the types of things you do in programs. Then you can build a final program simply by ENTERing each of the subroutines you want to have as part of a program, then write a small program which calls them all! It may make your programming job somewhat easier.

Here is a set of command lines you can type into the machine to try the LIST/ENTER function if you wish:

```
10 REM THIS IS PART 1
100 REM THIS LINE WILL GET REPLACED
LIST "D:PART1"
NEW
```

Now nothing is in memory.

```
100 REM THIS IS PART 2
LIST "D:PART2"
NEW
```

Notice that this time the disk unit was specified only as "D:" instead of "D1:". This is accepted by ATARI DOS as the default drive number. If no number is specified, it believes you mean disk unit number 1. If you are using ATARI DOS-2, you will have to have a program called DUP.SYS on the diskette. Now type

DOS **RETURN**

The system will load in the DOS menu controller.

To see the names of the files currently on the diskette you are using, call up the file index function (also called the disk directory) by hitting the "A" key and pressing **RETURN** twice. Among the files present on the disk should be files called

PART1

and

PART2

which you have just written to the disk. This shows that the LIST function has placed these programs on the disk.

Notice the rest of the names of the files on the disk. Some of them have one or more letters or numbers shown in a separate right-hand column, such as "SYS" or other symbols. Your ATARI DOS manual will tell you that names of files are formed by a letter, followed by up to seven additional letters or numbers, with an

"extension" of up to three letters or numbers. Such a title, when specified from ATARI BASIC, might look like this:

```
LIST  "D:FILE1234.BAS"
```

where the D: says that the file is to be listed onto drive 1, and its name is to be FILE1234.BAS.

When you list the file index, the "." which separates the name from the extension does not show up. The purpose of the extension is simply to let you identify further the name of a file. For example, the extension SYS is usually used by DOS to specify a system file. You may use BAS to specify that it is a file to be used with BASIC, or ASM to say it is to be used with the assembler, and so on. If you are only working with BASIC, though, the DOS is not fussy and does not care whether an extension is used or not. The extension is allowed, however, for your convenience. You might even wish to use the extension to keep track of which version of your program you are working with as it is being developed and saved, such as:

```
NEW
10 PRINT "THIS IS FIRST TRY"
SAVE "D:VERSION.1"
```

If you wanted to try this, you would have to select the run-cartridge function of the DOS menu (B **RETURN**). If you are following the text, and trying things as you go, select the cartridge function now. The text will enter the DOS at various points to look at other functions as well.

The last command will produce a file in the disk directory named VERSION 1, which you will be able to call into the machine with the command:

```
LOAD  "D:VERSION.1"
```

Or, if you wish to RUN the program directly, you may do so by entering the command:

```
RUN  "D:VERSION.1"
```

Every time you tell ATARI DOS to write a file onto the disk, it will first look to see if the disk has a write-protect tab in place (cov-

ering the notch in the disk). If there is a piece of tape covering the notch, DOS will refuse to write on the disk and will cause an error.

DOS will then read the directory to see if there is already a file present by that name. It will also try to find out where there is some space available on the disk to store the new data. A number of things can happen at this point:

1. DOS may find that it cannot read the data on the disk. If you have installed a disk which has not yet been "formatted," DOS will find no data present. See your ATARI DOS manual for data regarding how to format the disk.

2. If DOS finds no file by that name, it creates a new directory entry for the file. Along the way, DOS will keep track of which sectors on the disk have been reserved for use by that file, and finally write the directory entry for it so that your programs can find it later.

3. If DOS finds a file with that name, it creates a new temporary file as described in step 2. Then, on completion of the temporary file, DOS will delete the old copy of the file with the duplicate name. The reason DOS will do this is that when you ask it again to read a file with this name, DOS must have one and only one file with this name on the disk so it will know which file you wish to use. Therefore, if you wish to keep a "backup" copy of an old file while the new one is being prepared, do not use the same name for the new file. (It is usually a good practice either to use a new name, or to use the DOS RENAME-FILE function (E **RETURN**) to name a file as a backup).

NOTE: It is not the purpose of this book to repeat material available in the ATARI DOS manual. The information presented here is intended to help you use DOS functions in your programs.

Getting back to the earlier example, demonstrating the LIST and ENTER functions, assuming that you have files called PART1 and PART2 on disk unit 1, type the following commands:

```
NEW
ENTER "D:PART1"
LIST
```

```
ENTER  "D:PART2"
LIST
```

Thus you can see that the ENTER command did not disturb the program resident in memory, but, just as from the keyboard, a line with a duplicate line number in the second file replaced the original version of that same line number in the final version.

Now that some kinds of programs can be sent out to the disk and brought back in, other kinds of disk commands can be shown. The next program piece will be used for a number of purposes. First it will be a demonstrator for some of the disk commands. It will also be the program on which some disk work will be done.

The commands to be discussed here are OPEN, INPUT, PRINT, and CLOSE. The program will be used to read a file from the disk (in this case the program itself), and write the file back someplace else. In the first example, the file will be written to the screen. In the second version, it will be written into a different disk file.

You may recognize this as a primitive version of the DOS COPY-FILE function. Because it is done in BASIC, it will be slower than the same function performed directly by DOS. However, the purpose of this program is to learn how BASIC can work with DOS to perform certain functions. Notice that it will only function with program files (or text files . . . we will show these later in this book) which have been LISTed to the disk.

The second version of the program will function with any kind of ATARI DOS file (unprotected, of course). As in other program examples, these have been kept as short as possible, with comments following the program to explain what is being done. Here is the program:

```
10  DIM A$(120)
20  OPEN#1,4,0,"D:INFILE.LIS"
80  TRAP 1000
100  INPUT#1;A$
150  PRINT A$
180  GOTO 100
1000  CLOSE#1:END
```

Put this program on the disk by the command:

```
LIST "D:INFILE.LIS"
```

Now issue the command:

```
RUN
```

The result of the program RUN is to perform the exact same output as if you had issued the command LIST. (Issue the command LIST immediately to see the same result.)

What you have done here is to provide a program which can LIST, to the screen, files that have been placed on the disk by the LIST command. Here is how it was done:

Line 10 says:

```
DIM A$(120)
```

This means to reserve a character string of a maximum of 120 characters to hold the input lines. ATARI BASIC can only accept a maximum of 120 (actually about 114) characters in a standard line. Therefore, 120 reserved spaces for the line are sufficient. When a LIST command is performed, the data that is sent to the output device (in this case the disk) is sent as a set of strings, terminated by an end-of-line character. This, then, is what the program expects to read and is also the reason it worked only with LISTed programs or text files as mentioned earlier.

Line 20 says:

```
OPEN#1,4,0,"D:INFILE.LIS"
```

This means that an IOCB (see Chapter 1) is to be reserved for receiving data from a disk file named INFILE.LIS on disk unit number 1. The ATARI operating system will be managing the input and ouput of data, and will know that any INPUT commands issued to device #1 will be referencing the next available data in the file named here.

Line 80 says:

```
TRAP 1000
```

This means that when an error occurs, the program is to go to line 1000. There will definitely be an error *when the input file runs*

out of data. The program does not have any statements in it which would look for a last program line or anything like that. Therefore, the program, as you will see, keeps reading data until it hits the end-of-file. The trap statement prevents any error from being printed and provides a smooth way out of the program.

The statement in line 1000 is not really necessary (CLOSE#1) because ATARI DOS automatically closes all open files when it completes a program, whether by finding an error or any normal end. However, it is good practice.

Line 100 says:

```
INPUT#1;A$
```

This takes the program lines (strings) one at a time as input from the disk file.

Line 150 says:

```
PRINT A$
```

This prints the strings (program lines) to the screen.
Line 180 says:

```
GOTO 100
```

This simply forms an endless loop. When you want to run this program again, you would type:

```
NEW
ENTER "D:INFILE.LIS"
```

Then type:

```
RUN
```

However, this program is not very versatile. You may wish to make it more useful before deciding to save it for future use. How about adding the capability to ask the user which file name he wishes to have listed from the disk to the screen? The following is a set of changes and additions to the preceding program. The entire program has been modified and presented here, and the changes are explained following the program.

```
10 DIM A$(120)
12 DIM B$(20),NAME$(20)
15 PRINT "WHICH FILE IS TO BE LISTED?"
16 INPUT B$
18 NAME$ = "D:"
19 NAME$(3) = B$
20 OPEN#1,4,0,NAME$
80 TRAP 1000
100 INPUT#1;A$
150 PRINT A$
180 GOTO 100
1000 CLOSE#1:END
```

Line 12 was added to save space for two new strings. The first one is B$, which line 16 will pick up from the user.

Line 18 sets up the original value of NAME$ to be "D:", and line 19 adds the text provided by the user to the string.

Line 20 was changed from:

```
OPEN#1,4,0,"D:INFILE.LIS"
```

to:

```
OPEN#1,4,0,NAME$
```

This makes a user-defined file-name string available to the ATARI BASIC OPEN command.

If you RUN the modified version and enter:

```
INFILE.LIS
```

When the program asks which file, you will get the same result as earlier.

Notice that this is the old version of the program that you are listing. If you want to store the new version of the program for future use, type:

```
LIST "D:INFILE.LIS"
```

Or, you can save it under any other name you wish to use.

The original program may undergo another slight modification which may help you save some time in your program develop-

ment. If you recall earlier in this chapter, to list the directory of the disk, you were told to enter DOS. Remember that it took some time to load the DOS files, and you were left in the DOS system, rather than BASIC.

It is possible to list the directory of a disk, without ever leaving BASIC, by using the following program. Again, this program is a small modification of the first version of the program you wrote earlier.

```
10 DIM A$(120)
20 OPEN#1,6,0,"D1:*,*"
80 TRAP 1000
100 INPUT#1;A$
150 PRINT A$
180 GOTO 100
1000 CLOSE#1:END
```

The only change to this program is in line 20. The new version replaces the 4 with a 6 in the OPEN statement and respecifies the control string. The 6 says that the specified file is to be opened for directory read operations only. The control string, D1:*.*, is a "wildcard" specification, which means that the directory is to be searched for every file it has, and the names of the files are to be available as character strings for input from a program. This program then outputs the same information which the file directory function of DOS will output, including the total number of free sectors on the disk.

If you wish to save this program for future use, a name such as DIRLIST might be appropriate, so SAVE the file as:

```
SAVE "D:DIRLIST"
```

Then, whenever you need to use this program from BASIC, you can issue the command:

```
RUN "D:DIRLIST"
```

However, note that the RUN command will erase any program you may be presently working on in memory. This is one of those times when you might want to take advantage of the LIST and

ENTER commands instead. Let's see how. Modify the program line numbers to be:

```
32000 DIM A$(120)
32001 OPEN#1,6,0,"D1:*,*"
32002 TRAP 32006
32003 INPUT#1;A$
32004 PRINT A$
32005 GOTO 32003
32006 CLOSE#1:END
```

Now, instead of SAVEing the file for future use, LIST the file to the disk for future use:

```
LIST "D:DIRLIST.LIS"
```

If you are in the middle of working on a program, you might want to list the directory on the screen to see if you have already put a program by a particular name on this disk, or simply see if the current disk has enough room on it to hold your program. By issuing the following commands, you can call in this program segment and use it without disturbing the program you are working on. (Of course, there must be no duplicate line numbers, or the ENTERed program will replace them with its own.)

```
ENTER "D:DIRLIST.LIS"
GOTO 32000
```

This will list the directory of the disk in unit 1, and exit leaving your program intact.

Although this technique has performed the selected function, it does leave a little bit of a mess behind. In other words, the piece of a program which produced the directory is now a part of your program. If you should decide that your program file has no room for this function, you would want to delete it. The ATARI BASIC cartridge does not provide for any built-in block edit functions, such as renumber, or delete, and so on. Therefore, using the functions you just learned, you can add to the simple directory list program, a set of statements which will cause the segment to erase itself immediately after it executes! Here is the program, with the modifications included:

```
32000 DIM A$(120)
32001 OPEN#1,6,0,"D1:*,*"
32002 TRAP 32006
32003 INPUT#1;A$
32004 PRINT A$
32005 GOTO 32003
32006 CLOSE#1
32007 OPEN#1,8,0,"D1:DELETE,ME"
32008 FOR N=32000 TO 32012
32009 PRINT#1;N
32010 NEXT N
32011 CLOSE#1
32012 ENTER "D1:DELETE,ME"
```

The added lines are 32007 through 32012. When you wish to delete a line in ATARI BASIC, you must enter the line number and no-contents (a blank line with that line number).

This program piece performs an OPEN command on a file in write mode (the number 8 in the position shown in line 32007). Then it outputs to the file a set of blank lines each preceded by the line numbers you want deleted from your program. This program, then, as modified, will be ENTERed. You run the segment by a GOTO 32000, and it will list the directory of the disk, and then delete itself from your program. All of this is done without entering DOS directly.

Note that the disk with the directory being listed *must not be write-protected,* since the program piece needs to write to the disk to generate the file named DELETE.ME. However, if you wish, you can write the file called DELETE.ME separately, as well as the file DIRLIST.LIS. In that case, only line 32012 would have to be added to the original version of the program, and not the lines which write DELETE.ME.

You now have two DOS tools, one for file listing of already LISTed files, another for directory listing while still in BASIC. Now you will see how the file listing program can be generalized to handle file copying, even of files which have been produced by other means than LIST. Here is a complete version of a program which will ask the user the name of the file to copy from and the

name of the file to copy to. Notice that the main difference between this and the original is in added user communication.

```
12 DIM NAMEA$(20),NAMEB$(20)
14 PRINT "SPECIFY SOURCE DEVICE AND
   FILE"
15 PRINT "SUCH AS D1:SOMENAME.BAS"
16 INPUT NAMEA$
17 PRINT "SPECIFY DESTINATION DEV AND
   FILE"
18 PRINT "SUCH AS D1:NEWNAME.BAS"
19 INPUT NAMEB$
20 OPEN#1,4,0,NAMEA$
22 OPEN#2,8,0,NAMEB$
80 TRAP 1000
100 GET#1,M:PUT#2,M:GOTO 100
1000 CLOSE#1:CLOSE#2:END
```

In lines 20 and 22, one IOCB has been opened for reading the source file, and another has been opened for writing the destination file. When end-of-file is reached on the source, both files are closed. The second file should then be an exact copy of the first. If the source file was stored on disk by a SAVE command, then the copy of that file can be LOADed or RUN exactly like the source.

In line 100, instead of INPUT and PRINT, the commands GET and PUT are used. GET reads a single byte value from the input file; PUT writes that single byte value into the destination file. These commands are condensed onto a single line in an effort to speed up the program operation a bit.

HOW TO SAVE DATA

So far in this chapter, you have seen program examples which read and write disk files one line at a time, and programs which read and write files one character at a time. Each of the programs shown thus far has had no concern for the contents of those files because their purpose has only been to copy the contents of the

file from one place to another. The program examples that follow will show you how to use the disk as a data storage device for use during your programs.

The exercise that will be used to demonstrate data storage is a program for generating DATA statements. You will often find programs given in magazines and books which you may wish to try out on your machine. This program will help you in generating any DATA statements that might be needed in the program.

When you are entering DATA statements, it is often critical that each of the numbers or characters be entered exactly as designed. Sometimes the items in the DATA statements turn out to be machine instructions. If they are not entered precisely, the computer may, as it is said, go off into "never-never land." Thus, this program will help you get things right the first time.

This program is actually a complete series of three individual programs, each of which could be separated and used for other purposes. The first part of the program accepts raw data from you and builds a disk file containing essentially each of the data entries on a separate line. This kind of data file may be useful to you for certain kinds of programs.

The second part of the program is an *editor*. It will OPEN and READ the raw data file, asking if each item is correct, or if you wish to change, insert, or delete any item.

The third part of the program takes the corrected data file and uses it to generate DATA statements (with many data items per program line) for use in your BASIC program. You will be able to process this final file using an ENTER command, or to LIST the contents of any of the files used here using the file listing program developed in the earlier part of this chapter.

The individual programs are themselves going to be made up of small, possibly useful pieces. Each step in the building of the programs will be explained as it is encountered.

This program accepts the data items from you one at a time. You will be asked to enter each data item, then to press **RETURN**. When you are finished entering data, you then press **RETURN** again, with nothing at all entered on the input line. The program will consider this as the end of the data entry phase and will close the data file.

Entering Data

The first thing that must be arranged is a way to enter the data. The keyboard will be the input device. The following program piece will run all alone. It OPENs a control block (IOCB) for data input from the keyboard, and outputs to the screen the number that represents the key the user touches.

The purpose of this program piece is to let you decide which key values are to be accepted for the program. You will see in the large program that only certain ranges of key values will be accepted. This program lets you see why the number range shown was chosen.

```
20  OPEN#1,4,0,"K:"
25  REM PROVIDE A WAY TO GET DATA
    WITHOUT RETURN KEY
26  REM IOCB #1 BECOMES INPUT PATH FOR
    THE KEYBOARD
700  GET#1,M
710  REM READ CHARACTER FROM KEYBOARD
720  PRINT M
730  GOTO 700
```

RUN this program. Now touch the A key; it prints the value 65. Touch the Z key; it prints the value 90. Touch the 0 key; it prints the value 48. Touch the 9 key; it prints the value 57.

The values between A and Z are within the range of 65 to 90; the values for keys between 0 and 9 are within the range of 48 to 57. (You may touch other keys to confirm this.) You may decide later to add other individual values of keys to the recognition program. This program piece will help you select which other values will be used. If you wish to save this program piece, call it "KEYS." Save it to disk by typing:

```
SAVE"D:KEYS"
```

The next step is to add to this program the ability to tell the user if an acceptable key has been touched. Any key in the two selected ranges should be accepted directly. The user should be able to use the backspace key to correct errors during data entry.

Finally, the **RETURN** key should be accepted to indicate the end of the data entry. The other requirement is that the user should be told whether or not the key was accepted by the program.

Here is a modified version of the key-reading program which will do what is needed. If the key is acceptable, the key will be printed on the screen. If it is not either within the range mentioned, or not a **RETURN** key, it will be rejected and the buzzer will sound. (Either the buzzer within an ATARI 400/800 or the speaker of the television for one of the other ATARI computers will sound.)

In addition to the ability to accept the keystrokes and to say whether they are acceptable, this modification adds the ability to save the data in a string.

For this, as well as most of the other examples in this book, the REM statements are contained on separate lines. You do not have to enter them if you wish to minimize the amount of typing in the process of learning the program techniques.

```
10 DIM A$(120)
15 REM SAVE SPACE FOR AN INPUT DATA
   STRING
20 OPEN#1,4,0,"K:"
25 REM PROVIDE A WAY TO GET DATA
   WITHOUT RETURN KEY
26 REM IOCB BECOMES THE INPUT PATH FOR
   THE KEYBOARD
40 N=0:REM START COUNT OF CHARACTERS
650 GR,0:N=0:A$="":REM CLEAR THE
    SCREEN, NO DATA YET
660 PRINT:PRINT "ENTER A DATA
    ITEM":PRINT
700 GET#1,M
701 REM READ A CHARACTER FROM THE
    KEYBOARD
710 IF M=155 THEN 800
711 REM TEST FOR RETURN KEY TO STOP LINE
```

```
720 IF M=126 THEN 900:REM PROCESS THE
    BACKSPACE KEY
730 IF M < 31 OR M > 122 OR ((N+1)>120)
    THEN PRINT CHR$(253);:GOTO 700:REM
    SOUND BEEP
760 PRINT CHR$(M);
765 N=N+1:REM COUNT INPUT CHARACTERS,
    120 MAX
770 A$(LEN(A$)+1) = CHR$(M):GOTO 700
771 REM ADD TO CURRENT STRING LENGTH
800 FOR C=1 TO 300:NEXT C:REM DELAY
830 GOTO 40
900 N=N-1:IF N<=0 THEN 650
901 A$(N) = A$(N,N):PRINT
    CHR$(126);:GOTO 700
```

This program piece then only performs the data entry and checking. You may RUN this part if you wish just to see how it handles key entries. The comments within the program segment explain what it is doing.

Lines 900 and 901 process the **DELETE/BACK S** key (CHR$(126)) so that data can be corrected as it is entered. Line 900 moves the string pointer back by one count and returns the program to line 650 if the count reaches zero. Line 901 shortens the string, moves the display cursor back one position, then goes back to line 700 to obtain another character.

The program must now be modified to actually put away the data being entered into a file called RAW.DTA on the disk. This is done by adding the following lines:

```
30 OPEN#3,8,0,"D:RAW.DTA"
41 NR=0:A$="":REM SET START VALUES
770 A$(LEN(A$)+1)=CHR$(M):GOTO 700
771 REM ADD TO THE CURRENT STRING
    LENGTH
800 PRINT#3;A$:NR=NR+1
801 IF LEN(A$)=0 THEN 2005
805 A$="":REM LENGTH=0
2005 CLOSE#3
```

Line 30 sets up a place where the data can be stored. A new or existing disk file named RAW.DTA is opened for output only. Lines 41 and 805 each set the initial string length to zero. New characters are added to the end of the string each time line 770 is performed. When the string is finished (when you press **RETURN**), the string is written to the output device, which is the RAW.DTA file. If the string is empty, with a **RETURN** only, the file is finished, and so is this program segment.

A file produced by this program can be listed to the screen using the file listing program developed earlier in this chapter. You may choose to try this program at this time. Be sure to SAVE your work, just in case of any problems. A suggested way to save this part of the program is:

```
LIST "D:MAKERAW.LST"
```

You can bring it into the machine again later with the command:

```
ENTER "D:MAKERAW.LST"
```

The ".LST" was chosen so that when you look at the disk directory, you will remember that this program was LISTed to disk. Before you can RUN this kind of program, you must first ENTER it into the machine.

Editing Data

The second part of the program allows you to check each of the data items. The options are to accept the entry as it stands (by just pressing **RETURN**). Or you may change the data item and press **RETURN** to accept the changed value. You may insert a new data item above the one you are currently displaying by using the **INSERT** key, or you may delete the data item you are displaying by using the **DELETE** key.

This is not intended to be an advanced function line editor. Its primary purpose is to show you how data items may be read or written from or to a disk file. The lines of the program will be explained individually, then the entire editor program will be gathered together in a complete listing.

The program being developed here assumes that you have

already created a file using the MAKERAW.LST program. If you
have not yet done so, you may wish to do it now.

First, type:

```
NEW
ENTER  "D:MAKERAW.LST"
RUN
```

Then type:

```
100 RETURN
300 RETURN
500 RETURN   RETURN
```

Now, you will have a file named RAW.DTA on your disk.

The editor program segment can stand alone if you wish. It
was written that way, but by using the ENTER command, the
editor program can be combined with the MAKERAW program to
form a complete package. This is done later in the chapter. For
now, however, here is the stand-alone editor. Before you start,
type NEW.

The program line

```
225 OPEN#2,4,0,"D:RAW.DTA"
```

opens the RAW.DTA file for reading using IOCB #2. It assumes
that the file named RAW.DTA is already in existence, as it will be
if the MAKERAW program has already been run.

```
205 TRAP 2005
```

If this program is being run alone and there is a disk error, the
program assumes it couldn't find a file by that name (or maybe
the disk was not on, or the door was open).

```
2005 CLOSE#2:CLOSE#3
```

These statements provide a way to gracefully end when the
input file reaches an end-of-file indicator. When there are no more
statements to process in the file called RAW.DTA, the program
will close both the input and the output files.

For the next part of the program, to edit the data, you must
present it to the user, then ask him to accept it by pressing

RETURN, or insert another entry above this one (**INSERT**), or delete this entry (**DELETE**).

Some of the lines used in the earlier key-read part of this program will be repeated here. You may, if you wish, collect similar lines into a subroutine.

```
10 DIM A$(120),B$(120):REM SPACE FOR
   SOURCE DATA
20 OPEN#1,4,0,"K:"
30 REM FIX FOR DIRECT KEY READ
40 N = 0:REM COUNT CHARACTERS
41 NR = 0:REM COUNT OUTPUT RECORDS
42 INSERT = 0:REM NOT IN INSERT MODE
420 OPEN#3,8,0,"D:EDITED.DTA"
421 REM MAKE A PLACE FOR THE EDITED
    DATA
430 R = 1:NR = 0
435 REM START WITH FIRST SOURCE RECORD
445 GR.0:PRINT:PRINT "SIMPLE EDITOR-
    OUTFILE RECORDS:";
446 PRINT NR:PRINT:PRINT:PRINT
456 PRINT "SOURCE RECORD # ";R:PRINT
457 IF INSERT = 0 THEN INPUT #2;B$:R =
    R+1:REM COUNT A NEW SOURCE RECORD
458 PRINT B$:PRINT:REM PRINT SOURCE
    FILE AND BLANK LINE
459 INSERT = 0
460 PRINT:PRINT "DELETE, INSERT,
    RETURN?"
```

Line 460 presents the user with his possible key entries at this point. Any other keystroke should be ignored. (You might also want to add beep on bad key.)

```
465 GET#1,M
466 REM READ THE KEYBOARD
470 IF M=155 THEN 600
```

Line 470 processes the return key as an accept-source. This means to move the data to the output file and return to read another source record.

```
480 IF M=126 THEN 445
```

In line 480, 126 is the value of the **DELETE/BACK S** key. In this mode, the **DELETE** key says to delete the source record. Therefore, the process will be required to simply go back for the next source.

```
490 IF M=157 THEN N=0:GOTO 680
```

Line 490 processes the **INSERT** key. This means accept new data to be inserted ahead of the currently displayed record. For example if record #2 is being displayed, the insert will install the new record just in front of record #2, and behind record #1. The order will then be 1, new, 2, . . . last. If **INSERT** is used another time, then the next record to be inserted will also be inserted ahead of record #2 if it is still displayed as the current record. Thus, the order of the EDITED.DTA file will be 1, new, new2, 2, . . . last when many records are inserted.

```
500 GOTO 465
501 REM IF NOT INSERT, DELETE, OR
    RETURN, THEN IGNORE THE KEYBOARD
600 PRINT#3;B$:NR=NR+1: GOTO 445
601 REM BUILD OUTPUT RECORD FROM INPUT
    SOURCE RECORD
680 GR.0: PRINT "INSERT ABOVE SOURCE
    RECORD # ";R:PRINT
681 REM CLEAR SCREEN AND PRINT USER
    INSTRUCTIONS
```

This next part is similar to the input program given earlier. It must accept a new record and put it away in the destination file.

```
690 N=0:A$="":REM NO CHARS, IN REC, YET
700 GET#1,M
710 IF M=155 THEN 800:REM RETURN
```

```
720 IF M=126 THEN 900:REM BACKSPACE
730 IF M<31 OR M>122 OR ((N+1)>120)
    THEN PRINT CHR$(253);:GOTO 700:REM
    BAD CHARACTER
760 PRINT CHR$(M);
765 N=N+1
770 A$(LEN(A$)+1) = CHR$(M):GOTO 700
771 REM ADD TO STRING AND GET ANOTHER
    CHARACTER
800 PRINT#3;A$:NR = NR+1
801 REM MOVE NEW STRING INTO OUTPUT
    FILE
805 A$ = "":REM AFTER WRITING, SET
    LENGTH OF A$ TO ZERO
820 INSERT = 1
830 GOTO 445
831 REM REPRINT TITLE AND CURRENT
    SOURCE RECORD AFTER INSERT
    COMPLETED
900 N = N-1:IF N<=0 THEN 680
901 A$(N) = A$(N,N):PRINT
    CHR$(126);:GOTO 700
902 REM PERFORM EDITOR BACKSPACE
    FUNCTION
```

A stand-alone version of this program has been collected together here so that you can check your work if you have been entering it as you read the text:

```
1 REM STAND-ALONE SIMPLE EDITOR
10 DIM A$(120),B$(120)
20 OPEN#1,4,0,"K:"
40 N = 0
41 NR = 0
42 INSERT = 0
205 TRAP 2005
225 OPEN#2,4,0,"D.RAW.DTA"
420 OPEN#3,8,0,"D.EDITED.DTA"
430 R = 1:NR = 0
```

```
445 GR.0:PRINT:PRINT "SIMPLE EDITOR-
    OUTFILE RECORDS:";
446 PRINT NR:PRINT:PRINT:PRINT
456 PRINT "SOURCE RECORD # ";R:PRINT
457 IF INSERT = 0 THEN INPUT #2;B$:R =
    R+1
458 PRINT B$:PRINT
459 INSERT = 0
460 PRINT:PRINT "DELETE, INSERT,
    RETURN?"
465 GET#1,M
470 IF M = 155 THEN 600
480 IF M = 126 THEN 445
490 IF M = 157 THEN N = 0:GOTO 680
500 GOTO 465
600 PRINT#3;B$:NR = NR+1:GOTO 445
680 GR.0:PRINT:PRINT "INSERT ABOVE
    SOURCE RECORD #";R:PRINT
690 N = 0:A$ = ""
700 GET#1,M
710 IF M = 155 THEN 800
720 IF M = 126 THEN 900
730 IF M < 31 OR M > 122 OR ((N+1)>120)
    THEN PRINT CHR$(253);:GOTO 700
760 PRINT CHR$(M);
765 N = N+1
770 A$(LEN(A$)+1) = CHR$(M):GOTO 700
800 PRINT #3;A$:NR = NR+1
805 A$ = ""
820 INSERT = 1
830 GOTO 445
900 N = N-1:IF N < = 0 THEN 680
901 A$(N) = A$(N,N):PRINT
    CHR$(126);:GOTO 700
2005 CLOSE#2:CLOSE#3:END
```

LIST this program to disk as EDITRAW.LST. It will be referenced later by this name.

REVIEW OF CHAPTER 3

1. Programs can be saved to or loaded from the disk system. A program that is SAVEd is stored in a special form. When that program is LOADed or RUN, ATARI BASIC performs an automatic NEW command, erasing any other program present in memory.
2. Instead of SAVEing a program, it may be LISTed to the disk system, in pieces if desired. This allows the ENTER command to be used to build up complete programs from the individual pieces that you have already tested and used previously.
3. The DOS OPEN command can be used to prepare a disk file for reading or writing, or can be used to access the directory on the disk. Records or individual characters can be either read or written to the disk file, thereby allowing you to create or modify disk records. The BASIC commands GET and PUT are used to retrieve or write individual characters. The commands INPUT and PRINT are used to read or write complete strings as disk records.
4. Using the disk commands directly, you can produce programs directly in ATARI BASIC that do the things which are normally only done while DOS is active. This allows you to stay exclusively in ATARI BASIC and make the program do some of what you might do "by hand." In addition, you can form your own menu-driven routines for certain activities, with user outputs suiting your own tastes.

CHAPTER 4

More Suggestions About Disk Operations

The last chapter explained most of the DOS commands which can be directly called by name from ATARI BASIC. Specifically, those for getting characters and getting records (GET, INPUT), and those for placing characters and placing records (PUT, PRINT).

It also showed that the OPEN command with an IOCB number is needed before a disk file is used, to prepare the system to communicate with that data file on the disk (or to create one if it does not already exist).

Now that you have been introduced to an ability to list a directory directly from BASIC, and to copy a file directly from BASIC, you may wish to be able to delete a file and rename a file also.

Before explaining *how* to do either of these, you might want to know *why*. The example shown at the end of Chapter 3 shows a very simple editor which takes one line of data at a time from a source file called RAW.DTA and rewrites each line, or a modified version of that line, or a new line into a new file called EDITED.DTA.

Since it is such a simple editor, there is certainly a chance that you may want to repeat the process a number of times, checking

out the records and making sure they are in the right sequence, or making additional changes.

If you wanted to use the editor more than one time on the data file, you might have to go into DOS and perform the following sequence:

1. Delete RAW.DTA.
2. Rename EDITED.DTA to RAW.DTA.
3. Go back to BASIC.

This sequence would be required because the program, as it is written, always looks for the file named RAW.DTA, and always produces its output called EDITED.DTA.

There are two possible solutions for this problem. In solution 1, which you may try if you wish, you could have the program ask, "What is the name of the input file?" This would require that you add a DIM statement to define a new string to hold the input file name, adding the statement which requested the name, then change line 225 to read:

```
225  OPEN#2,4,0,D$
```

In addition, you would have to add a statement to ask the user, "What is the name of the output file?", a DIM statement for a string for the output file name, and a change in line 420 to read:

```
420  OPEN#3,8,0,D$
```

Now these changes are quite acceptable, and do generalize the program somewhat. But the program still does not allow you to effectively ask the question, "Do you want to rerun the edit?"

But even if you assume that you have already added the statements needed to make the program general regarding the file name questions, you would want this sequence to edit the *revised* data, and NOT the *original* data again.

You could, of course, make the program GOTO a position in the program that would ask the user what file name to use as input and which one to use as output. But this is really not desirable, since the edit process should be somewhat automatic. In addition, if this kind of approach was used, you would have to remind the user what names he or she used for input and for

output. Otherwise, the user might not remember, and might ask for a name that did not exist, or by accident, wipe out the wrong file. The necessary error traps might take up a lot more program space than other alternate methods.

In the process of making the program automatic, let's assume you have made changes to the program to use D$ to generate and hold the input and output file names for the edit program. To re-edit the data, you must do the sequence suggested earlier. The names used here will match those string names that were just chosen:

1. Delete the RAW.DTA file from the disk.
2. Rename the file now named EDITED.DTA to RAW.DTA.
3. Then go through the editor again.

First, the program piece will be listed. Then the process it uses will be explained. All of these lines are to be added to the EDITRAW.LST program.

```
12 DIM INNAME$(20),OUTNAME$(20)
15 DIM FILE$(20),D$(20)
200 GR.0:PRINT:PRINT "OPEN WHAT FILE
    NAME ON D1?"
205 PRINT:TRAP230:POKE195,0
212 D$ = "D1:"
215 INPUT FILE$
220 D$(4) = FILE$
221 INNAME$ = FILE$
225 OPEN#2,4,0,D$
230 IF PEEK(195)<>0 THEN 2005
260 PRINT:PRINT
270 PRINT "WHAT NAME FOR OUTFILE ON
    D1?"
280 PRINT
290 INPUT FILE$
295 D$ = "D1:"
300 D$(4) = FILE$
301 OUTNAME$ = FILE$
420 OPEN#3,8,0,D$
```

```
430 R = 1:NR = 0
2005 CLOSE#2:CLOSE #3
2010 D$ = "D:"
2020 D$(3) = INNAME$
2030 XIO 33,#2,0,0,D$:REM DELETE OLD
     RAW DATA
2040 D$ = "D:"
2050 D$(3) = OUTNAME$
2060 D$(LEN(D$)+1) = ","
2070 D$(LEN(D$)+1) = INNAME$
2080 XIO 32,#2,0,0,D$:REM RENAME EDITED
     , RAW
2090 GR.0:PRINT:PRINT "DO YOU WANT TO
     RERUN THE EDIT?":PRINT
2100 GET#1,M
2120 IF CHR$(M) = "Y" THEN 2140
2130 IF CHR$(M) = "N" THEN PRINT
     "CLOSING FILES, DONE":END
2140 TRAP 230
2150 D$ = "D:"
2160 D$(3) = INNAME$
2170 OPEN#2,4,0,D$
2180 D$ = "D:"
2190 D$(3) = OUTNAME$
2200 OPEN#3,8,0,D$
2220 GOTO 430
```

Lines 12 and 15 of this program provide the extra string names which would be needed to generalize the edit program. Lines 200–430 ask you what to do at the program start. Line 2005 closes both the input and the output files. Lines 2090 through 2130 communicate with the user, asking if all is done, or if another try is to be made. Lines 2010 through 2030 are used to build up a string which may be used by the ATARI BASIC XIO statement. This statement is used here since there is no statement called "DELETE."

The XIO statement is one that is used to perform certain special functions in ATARI BASIC. The number that immediately follows the XIO defines which command is to be performed. The 33 used

in line 2030 is the delete-file command number. Following the 33
is the IOCB definition (#2), specifying which IOCB to use during
this command. Sometimes the system needs a small amount of
workspace, and this defines what is needed. The next two posi-
tions in the XIO commands are what are called the AUX1 and
AUX2 locations. These, you will see, are most often specified as
a value of 0. Only certain of the commands, such as OPEN,
require further specification data. (For instance, *how* do you want
the file to be OPENed?) Neither the delete-file nor the rename
commands need anything but 0 here.

The last item in the XIO command is the string which describes
where the file is located and its name. Lines 2010 through 2030
are used to build up the final contents of the string to be fed to
this command. The leftmost part of the string is "D:", which, in
this case, assumes disk unit 1. The next part of the string is
INNAME$. If the name is RAW.DTA, then the resulting D$ will be
"D:RAW.DTA".

Finally, for the delete function, line 2030 is executed. This line
says (when the string is interpreted):

```
2030 XIO 33,#2,0,0,"D:RAW.DTA"
```

but is really written as:

```
2030 XIO 33,#2,0,0,D$
```

which deletes the original file named RAW.DTA from the disk,
assumed to be disk 1. The reason this was described in this way
was to emphasize that these XIO commands (and sometimes
other commands) do not need to be written with the string con-
tents fixed within the command. It is only necessary to give ATARI
BASIC the equivalent string for its use and it will perform the
same activity in either case. This lets you change the contents of
the string, while being able to write certain general-purpose
statements, as demonstrated here.

The next set of lines (2040 through 2080) complete the rename
function. Specifically, the structure of the rename command, as
given in the DOS manual, is:

```
XIO 32,#2,0,0,"D:OLDNAME,NEWNAME"
```

The statements in line numbers 2040 through 2080 build, then execute, this rename sequence.

A further note about the sequence building:

1. Line 2050 says D$(3) = OUTNAME$. This is OK because D$(1) is known to be "D" and D$(2) is known to be ":"; therefore, the string, starting at position 3 and onward, can be filled with OUTNAME$ for whatever length this name might be.
2. Line 2060 says, however, that the new position where the comma is to be added is at LEN(D$) + 1. This is one beyond the current length of the string. You see, you don't know how long the output name string is and must account for that somehow when the new string is being built. The same goes for the position at which the input file name starts.
3. Finally, line 2080 executes the rename function.

EDITOR PROGRAM—FINAL VERSION

The program that follows is a combination of the file creator (MAKERAW) and the editor program (EDITRAW). You can produce this program by first reading in both of those old files, then listing the combined program to the screen, a few lines at a time. Most of the line numbers remain the same as used before.

This final version of the editor program has menus and instructions. It also lets you define the names of both the input file and the output file. Both files must be on disk unit 1.

If you have followed the suggestions given earlier and have kept a copy of MAKERAW and EDITRAW as LISTed files, then you can begin to construct this program by typing:

```
NEW
ENTER  "D:MAKERAW.LST"
ENTER  "D:EDITRAW.LST"
```

Then, start to edit and add the rest of the lines to complete the program. Before you begin to add the other lines, delete line 30 from MAKERAW—a substitute for that line occurs later in the

program. When you have finished, SAVE this program to disk as EDITOR.BAS.

```
1 ERRSAVE = 195
2 REM DEFINE WHERE SYSTEM SAVES ERROR
5 DIM TEMP$(120)
10 DIM A$(120),B$(120)
15 DIM FILE$(20),D$(20)
20 OPEN#1,4,0,"K:"
30 REM FIX FOR DIRECT KEY READ
40 N = 0: REM COUNT CHARACTERS
41 NR = 0:REM COUNT OUTPUT RECORDS
99 REM PRINT THE MENU
100 GR.0:PRINT: PRINT "SIMPLE
    EDITOR":PRINT
110 PRINT "1. EDIT OLD FILE":PRINT
120 PRINT "2. CREATE NEW FILE":PRINT
130 PRINT "3. QUIT"
140 PRINT: PRINT "PRESS 1, 2, OR 3 TO
    SELECT FUNCTION":PRINT
150 GET#1,M
160 IF M = 51 THEN 2005:REM 3 = QUIT
165 IF M = 50 THEN 250: REM 2 = NEW
    FILE
170 IF M = 49 THEN 200:REM 1 = OLD FILE
180 GOTO 150
200 PRINT "WHAT FILE NAME ON D1?"
205 TRAP 230:POKE ERRSAVE,0
212 D$ = "D1:"
215 INPUT FILE$
220 D$(4) = FILE$:REM BUILD A SPECIFIER
225 OPEN#2,4,0,D$
230 IF PEEK (ERRSAVE) = 136 THEN SOURCE
    = 0:GOTO 445
235 IF PEEK (ERRSAVE) <> 0 THEN 3000
239 REM SHOW AND TELL ANY UNKNOWN ERROR
240 SOURCE = 1:REM YES, READING SOURCE
245 GOTO 260
```

```
250 SOURCE = 0:REM NO, NO SOURCE FILE
260 PRINT: PRINT
270 PRINT "WHAT NAME FOR OUTPUT FILE ON
    D1?"
280 PRINT
290 INPUT FILE$
295 D$ = "D1:"
300 D$(4) = FILE$
420 OPEN#3,8,0,D$
430 R = 1
445 GR.0:PRINT:PRINT "SIMPLE EDITOR-
    OUTFILE RECORDS:";
446 PRINT NR: PRINT: PRINT: PRINT
450 IF SOURCE = 0 THEN 1800
456 PRINT "SOURCE RECORD # ";R:PRINT
457 INPUT #2;B$:R = R+1
458 PRINT B$:PRINT
460 PRINT:PRINT "Edit, DELETE, INSERT,
    RETURN?"
465 GET#1,M
466 IF M = 69 THEN 1000: REM E FOR EDIT
470 IF M = 155 THEN 600: REM RETURN KEY
480 IF M = 126 THEN 445: REM DELETE KEY
490 IF M = 157 THEN N = 0: GOTO 680:REM
    INSERT KEY
500 GOTO 465
600 PRINT#3;B$:NR = NR+1: GOTO 445
650 GR.0:PRINT:PRINT "ADDING A NEW
    RECORD AS OUTREC: ";NR+1
655 N = 0:A$ = "":REM NO CHARACTERS IN
    RECORD YET
660 PRINT:GOTO 700
680 GR.0:PRINT: PRINT "INSERT ABOVE
    SOURCE RECORD # ";R:PRINT
690 N = 0:A$ = "":REM NO CHARACTERS IN
    RECORD YET
700 GET#1,M
710 IF M = 155 THEN 800
720 IF M = 126 THEN 900
```

```
730 IF M<31 OR M>122 OR ((N+1)>120)
    THEN PRINT CHR$(253);:GOTO 700
760 PRINT CHR$(M);
765 N = N+1
770 A$(LEN(A$)+1) = CHR$(M): GOTO 700
800 PRINT #3;A$:NR = NR+1
805 A$ = ""
807 IF SOURCE = 0 THEN 445
810 GR,0:PRINT: PRINT "SIMPLE EDITOR-
    OUTFILE RECORDS:";NR:PRINT: PRINT:
    PRINT
820 PRINT "SOURCE RECORD # ";R:PRINT
830 GOTO 458
900 N = N-1:IF N< = 0 THEN 680
901 A$(N) = A$(N,N):PRINT
    CHR$(126);:GOTO 700
1000 A$ = B$:REM EDIT THIS RECORD
1010 GR,0
1020 PRINT: PRINT "EDITING SOURCE
     RECORD # "; R:PRINT
1030 PRINT "WILL BECOME OUTPUT REC #
     ";NR+1:PRINT
1040 POSITION 2,7:PRINT A$
1045 GOSUB 2500
1050 POSITION 2,7:PRINT A$(1,1);
1060 PRINT CHR$(30);
1070 N = 1:REM ON FIRST CHARACTER NOW
1100 GET#1,M
1110 IF M = 155 THEN B$ = A$:GOTO
     810:REM DONE, GET NEXT
1120 IF M = 31 THEN 1300
1121 REM FWD ARROW (SEE IF CHARS IN
     STRING, ELSE DON'T MOVE)
1130 IF M = 30 THEN 1400
1131 REM BACK ARROW (DON'T ALLOW MOVE
     BACK PAST 1ST CHAR)
1140 IF M = 157 OR M = 255 THEN
     1500:REM INSERT ONE SPACE (CTRL OR
     SHIFT + INS,)
```

```
1145 IF M = 254 THEN 1600:REM CTRL
     BACKS
1146 IF M = 126 THEN 1700:REM BACKSPACE
1150 IF M<31 OR M>122 THEN PRINT
     CHR$(253);:GOTO 1100
1151 REM TRAP ANY OTHER KEY THAN THOSE
     ALREADY USED
1152 REM (ALLOW ALL PRINTABLE ITEMS,
     THOUGH)
1155 IF ((N+1)>120) THEN PRINT
     CHR$(253);:GOTO 1100
1156 REM LIMIT RECORD SIZE TO 120 MAX
1160 PRINT CHR$(M);
1170 A$(N,N) = CHR$(M)
1180 N = N+1
1190 GOTO 1100
1300 IF (N+1)>LEN(A$) THEN PRINT
     CHR$(253);:GOTO 1100
1310 PRINT A$(N,N);:N = N+1
1320 GOTO 1100
1340 REM ADJUST FOR LINE BREAK IN PRINT
     OF THE LINE
1400 IF (N-1) = 0 THEN PRINT
     CHR$(253);:GOTO 1100
1410 N = N-1:PRINT CHR$(30);
1420 IF N = 38 OR N = 76 THEN PRINT
     CHR$(28);
1430 GOTO 1100
1500 TEMP$ = A$(N)
1510 A$(N) = " "
1520 A$(N+1) = TEMP$
1530 PRINT CHR$(255);
1540 GOTO 1100
1600 IF N = LEN(A$) THEN A$(N,N) = " "
     :GOTO 1100
1605 PRINT CHR$(254);
1610 A$(N) = A$(N+1):REM SHORTEN STRING
     BY 1
```

```
1620 GOTO 1100
1700 IF N<2 THEN PRINT CHR$(253);:GOTO
     1100
1701 REM DON'T ALLOW BACKSPACE BEYOND
     STRING START
1710 N = N-1:PRINT CHR$(126);:A$(N,N) =
     " ":GOTO 1100
1800 PRINT "Add new record, Done":PRINT
1810 GET#1,M
1820 IF M = 65 THEN 650
1830 IF M = 68 THEN 2005
1840 GOTO 1810
2005 CLOSE#2:CLOSE#3:END
2500 POSITION 2,14
2510 PRINT "TYPE OVER EXISTING TEXT,
     OR"
2520 PRINT "USE LEFT OR RIGHT CURSOR
     KEYS"
2530 PRINT "OR CTRL + INSERT TO OPEN A
     SPACE"
2540 PRINT "OR CTRL + BACKSP TO CLOSE A
     SPACE"
2550 PRINT "OR BACKSPACE TO CORRECT,
     THEN"
2560 PRINT "RETURN TO ACCEPT":RETURN
3000 GR.0:PRINT: PRINT "ERROR NUMBER =
     ";PEEK (ERRSAVE):END
```

DATA-STATEMENT PROGRAM

Thus far, you have seen two parts of the program sequence which was started at the beginning of Chapter 3. To remind you, this was to accept from you various *individual* data items. Then a second program was presented to allow you to view these items separately, and to change them if a change was needed.

Now that both of these parts have been presented, we can get to the part which puts them all together. This is to be the part that READs the data file, and puts all of the items into DATA state-

ments which can be ENTERed into a BASIC program. Again, these programs are not necessarily the only way of performing an operation. However, they do provide a way in which some of the ATARI BASIC statements may be combined to do useful work.

First, the program planning part. It is usually easiest to write the program correctly if you know ahead of time all of the steps that must be done to complete the job. Here are some of those steps:

   I. Open the EDITED.DTA file for reading, and open a file for writing the final DATA statements.
   II. Set up an output string of characters. This will be used to hold the completed DATA statement.
   III. Set up an input string of characters (whatever maximum you have allowed for in the program EDITOR.BAS). This will be used to read the input records, one at a time.
   IV. Ask the user what will be the starting line number for the DATA statements, and the size of the line number steps.
   V. Now begin a loop:
     A. Calculate a line number.
     B. Convert the line number to a string value using the STR$ function, and put it into the output string.
     C. Write the word "DATA" following the line number.
     D. Now go into another loop:
       1. Read an item from EDITED.DTA.
       2. See if the item length is less than the difference between the current position in the output string, and the end of available space in this string. If so, add it to the output string and go get another. If not, then write the output string to the disk and start a new output string. If the line you found still doesn't fit in the space of a new string, it must be too long and is, therefore, an error.
       3. Continue with the next data item; if end of file, stop.

Let's see how this would look in finished form:

```
1 REM DATAMAKR.BAS PROGRAM
10 DIM A$(110),B$(110),C$(5)
```

```
15 REM OUTPUT STRING, INPUT STRING,
   NUMBER
16 REM STRING AFTER LINE NUMBER
   CONVERTED,
18 BMAX=38:REM HOW MANY CHAR POSS IN
   OUT$
20 OPEN#1,4,0,"D:EDITED.DTA"
21 REM OPEN THE INPUT FILE
25 OPEN#2,8,0,"D:DATASTMT.LIS"
26 REM OPEN FINAL OUTPUT FILE
30 GR.0:REM CLEAR SCREEN
40 PRINT:PRINT"WHAT IS START LINE FOR
   DATA";
50 INPUT STRT
60 PRINT:PRINT"WHAT IS INCREMENT";
70 INPUT INCR
72 PRINT:PRINT
75 PASSFLAG = 1:REM SET VALUE TO START
80 FOR N=STRT TO 32767 STEP INCR
81 REM CONTINUOUS LOOP, BUT DONT LET
   NUMBER
82 REM GET ANY BIGGER THAN 32767.
90 C$=STR$(N)
91 REM CONVERT A NUMBER TO A STRING
   VALUE.
100 B$(1)=C$ :REM FILL IN UP TO 5
    CHARACTERS
120 B$(LEN(B$)+1)=" DATA ":REM PUT IN
    THE WORD.
121 REM ONE BLANK SPACE BEFORE AND
    AFTER THE WORD DATA, INSIDE THE
    QUOTES
125 ITEM=1:REM IF FIRST ITEM, THEN NO
    COMMA
128 IF PASSFLAG=2 THEN GOTO 140:REM
    DONT READ
129 TRAP 300:REM END OF FILE ON SOURCE
130 INPUT#1,A$:REM GET A DATA ITEM
```

```
135 PASSFLAG=1:REM WORKING ON FIRST TRY
    TO
136 REM INSERT THIS ITEM INTO A DATA
    STATEMENT
140 X=LEN(B$)+2
150 IF LEN(A$)> (BMAX-X-1) THEN 200
151 REM SEE IF THE NEW ITEM WILL FIT
152 REM IF GETS HERE, IT FITS,
    (INCLUDING
153 REM A PRECEDING COMMA).
156 IF ITEM=1 THEN X=X-1:GOTO 160:REM
    SKIP ","
157 B$(X-1)=","
160 B$(X)=A$:REM PUTS IT AWAY
170 ITEM=ITEM+1:GOTO 130
200 IF ITEM=1 THEN 2000:REM TOO LONG,
    ERROR
210 PASSFLAG=2
211 REM SECOND TRY TO FILE THIS ITEM,
    DO NOT
212 REM DO A DATA READ UNTIL SUCCESSFUL
    AT
213 REM FILING THIS ONE AWAY.
220 PRINT#2,B$
222 PRINT B$:FOR V=1 TO 300:NEXT V
223 REM DEBUG AID = PRINT TO SCREEN,
    THEN DELAY.
230 NEXT N:REM NEXT DATA LINE
250 PRINT:PRINT"ERROR-TOO MUCH DATA,
    NOT ENOUGH LINE NUMBERS TO HOLD IT
    ALL":GOTO 310
300 IF LEN(B$)>10 THEN PRINT B$
310 CLOSE#1:CLOSE#2:END
2000 PRINT:PRINT"RECORD TOO LONG TO FIT
    INTO DATA"
2010 PRINT:PRINT"STATEMENT-PROGRAM
    HALTED"
2020 GOTO 310
```

Save this program as DATAMAKR.BAS. If you use this program with your edited-data program, it will take, for example, the separate inputs which you might have generated as:

```
100
4
16
53
241
```

and so on and turn them into a set of DATA statements, which would begin:

```
31000  DATA  100,4,18,53,241
31015  DATA...
```

and so on, if you specified 31000 as the beginning line number and 15 as the increment.

The program will squeeze as many data items onto a line as will fit in the width you specify in line 18. Try a value of 38, for example. The data statements will then be no wider than the normal screen display.

The file you would produce would be suitable for ATARI BASIC to accept using an ENTER statement. This would add whichever lines you produce here to your program.

Instead of this approach to the final editing, you may decide to enter entire program lines, including the line numbers, into the RAW.DTA file and later, of course, the EDITED.DTA file. You may change your EDITOR.BAS program to accept a wider range of input characters to allow this. For example, instead of simply accepting the numbers 0–9, and the letters A–Z, you could expand the range of ATASCII values to include all characters from ATASCII value 32 (a blank space), through and including the value 122, which is a "z." This would allow you to accept normal program input lines into the RAW.DTA file. Then this raw data would be suitable to try to ENTER into the machine when completed. If you also revise the editor program segment to include these characters, then the EDITED.DTA file will also be in a position to be ENTERed directly. Thus, with these kinds of changes, you will be typing the program statements and DATA statements

directly, and will not need to final-process the items into DATA statements after all.

However, this program does provide a reasonable exercise in file handling and string handling, which was its main purpose.

LOCKING AND UNLOCKING FILES

While on the topic of functions that can be done directly from ATARI BASIC, there are two others you might find useful: LOCK and UNLOCK. LOCK, when applied to a disk file, prevents the DOS from erasing that file. When a directory list is done, the LOCKed files are those that appear in the listing with an asterisk (*) ahead of the name. UNLOCK reverses the process, removing the asterisk and letting you delete the file when desired, without issuing an error message.

To perform the LOCK function, you will issue the ATARI BASIC command line:

```
XIO 35,#2,0,0,"D:FILENAME.EXT"
```

where FILENAME.EXT stands for any file name and extension which you may wish to LOCK. The #2 can, of course, be any currently unused IOCB, from 1 to 5.

To UNLOCK a file, you will issue the command:

```
XIO 36,#2,0,0,"D:FILENAME.EXT"
```

Again, in place of the quoted string, you may substitute the name of a string you have designated. As demonstrated previously, you may build the strings one piece at a time.

RENAME PROGRAM

As a further example of the building of strings, the program that follows is an example of a stand-alone RENAME program. You may wish to add onto this program the ability to DELETE files, to LOCK files, to UNLOCK files, etc. This is just an example on which you can build larger programs if you wish.

```
1 REM STAND-ALONE RENAME PROGRAM.
3005 GR.0:REM CLEAR SCREEN
```

```
3010 PRINT "RENAME PROGRAM":PRINT:PRINT
3015 DIM Z$(1),OLDNAME$(12),
     NEUNAME$(12),RNAME$(40)
3020 PRINT "SPECIFY DRIVE NUMBER AND
     FILE NAME:"
3030 POSITION 2,6
3040 PRINT "DRIVE NUMBER (1 OR 2)"
3050 INPUT DRV
3055 IF DRV>2 OR DRV<1 THEN PRINT
     CHR$(253):GOTO 3030
3060 POSITION 2,8:PRINT "OLDNAME:"
3062 RNAME$ = "D"
3064 RNAME$(2) = CHR$(ASC("0")+DRV)
3066 RNAME$(3) = ":"
3068 REM FORM THE EQUIVALENT OF D1: OR
     D2:
3070 INPUT OLDNAME$
3075 IF LEN(OLDNAME$) = 0 THEN PRINT
     CHR$(253):GOTO 3060
3078 RNAME$(4) = OLDNAME$
3080 POSITION 2,10:PRINT "NEWNAME:"
3090 INPUT NEUNAME$
3100 IF LEN(NEUNAME$) = 0 THEN PRINT
     CHR$(253):GOTO 3060
3110 ND = LEN(RNAME$)+1:REM SAVE SOME
     WRITING LATER
3120 RNAME$(ND) = ",":RNAME$(ND+1) =
     NEUNAME$
3125 REM FINISHED FORMING THE
     INSTRUCTION STRING, DO IT
3130 XIO 32,#2,0,0,RNAME$
```

Notice in this program that the NEWNAME is spelled *NEU-NAME*. This prevents ATARI BASIC from accidentally erasing your program, which would happen if it recognized the keyword NEW embedded in a word. The only exception to this rule is when the keyword is used inside quote marks (" ") as part of a PRINT statement (Line 3080). Save this program by typing: LIST "D:RENAME.LST".

Something to Watch For:

ATARI DOS will *not* issue an error if you try to give the same name to more than one file on the disk. This means that if you ask DOS to do this, it will. This is not the immediate problem, though. When you ask ATARI DOS to RENAME that file again, or DELETE that file, or LOCK or UNLOCK, it will not be able to tell which one of those file names is the one you want it to operate on. In other words, whatever command you give, files with the same name will have the same action performed on *all* of them.

ATARI DOS has what is called a *wildcard* feature which allows multiple file names to be specified for a single command type. Those commands performed directly by DOS use this wildcard feature to perform the same function on any file name which matches the file specification the user provides. Therefore, if you specify exactly the equivalent of a file specification, and there are two on a disk of exactly the same name, ATARI DOS will perform the function on both.

If this should happen, there is a way to easily save the file whose name occurs *first,* starting from the top of the directory list. That is to enter DOS and perform the COPY file function specifying that file name. Only the first file with this name will be copied to a new, separate name you specify.

The second occurrence of this other file name is not as easy to retrieve. It will require the use of one of the available disk-editor programs, to go directly into the directory track and change the name on the disk itself. That effort is somewhat beyond the intended scope of this book and will, therefore, not be covered here.

How then can the user prevent this accidental duplicate name from happening in the first place? Well, most of the time when you look at a situation, then make a decision based on the things you see, it is often possible to somehow direct the machine to perform those same steps.

In particular, if you are to avoid making this mistake, you would probably go into DOS and list the directory of the disk. You would remember which names are there and could, therefore, pick a name that is not now being used. Then you would rename your current file to that new name.

You can add to the program such things as:

1. The steps required for the machine to look at the disk directory for you.
2. If any of the items already match the new name you have chosen.
3. Tell you that it can't perform the name change because that name is already taken.

This will not be performed here, but the required steps will be outlined, to give you a challenge of your own.

1. OPEN #2 for reading the directory of the target disk.
2. Set up two strings, one for the read of the directory itself, the other for holding a shifted version of the NEUNAME$. This is necessary because the first character of the directory list has at least 2 blank spaces preceding the first character of the file name.
3. Reformat the NEUNAME$ so that it can be compared directly to the directory entries.
4. Examine all directory entries to see if the name chosen is already there (don't allow user to choose a duplicate name).

Here is a program piece that will check the name for you. Before you begin to type this program, type NEW. This program is compatible with RENAME.LST, but can stand alone as a name-checking demo if you wish to try it that way first.

```
2 REM PATCH FOR CHECKING BEFORE
  RENAMING A FILE IF NEUNAME ALREADY
  THERE!
3 REM 2 ROUTINES - ONE TO SQUEEZE AN
  INPUT STRING
4 REM OTHER TO LOOP THRU THE DIRECTORY
  TO SEE IF THAT NAME IS THERE
5 GRAPHICS 0:PRINT "SPECIFY A FILENAME
  STRING"
10 DIM IN$(20),FILE$(8),EXT$(3),
   DIR$(20),SQ$(20)
20 INPUT IN$
30 GOTO 4000
```

```
3060 END
3110 END
3200 PRINT "ERROR NUMBER ";PEEK(195)
4000 REM REFORMAT USER STRING TO
     EXACTLY MATCH THE DIRECTORY ENTRY.
4010 SQ$="           ":REM 11 BLANKS
4020 FOR I=1 TO LEN(IN$)
4030 IF IN$(I,I)="." OR IN$(I,I)=" "
     THEN GOTO 4060
4031 REM FORM THE FILENAME STRING PART
     , TERMINATE IF BLANK OR PERIOO
     FOUNO
4035 IF I>8 THEN GOTO 4060:REM MAX 8
     CHARACTERS TO FILENAME.
4040 FILE$(I,I)=IN$(I,I)
4050 NEXT I
4060 REM NOW SEE IF THERE IS AN
     EXTENSION PART (SOMETHING AFTER A
     PERIOO)
4070 FOR I=1 TO LEN(IN$)
4080 IF IN$(I,I)="." THEN 4200
4090 NEXT I
4100 EXT$="   ":GOTO 4300:REM IF NO
     PERIOO,THEN NO EXTENSION ON NAME
4200 IF (I+1)>LEN(IN$) THEN GOTO 4300
4210 EXT$=IN$(I+1):REM LAST PART OF
     USER NAME MUST BE EXTENSION PART
4300 SQ$(3)=FILE$:REM FORM FIRST PART
     OF NAME
4310 SQ$(11)=EXT$:REM AOD EXTENSION
4311 REM NOW USER NAME MATCHES
     OIRECTORY ENTRY FORM, SO CAN SEE
     IF IT IS ALREADY THERE!
5000 OPEN #2,6,0,"D:*.*"
5010 TRAP 6000:REM TRAP END OF FILE
5020 INPUT #2,DIR$
5021 REM SHORTEN OIR$ TO 11 CHARACTERS
5030 OIR$(13)=OIR$(13,13)
```

```
5031 REM SHORTEN DIR$ TO 13 CHARACTERS
5040 IF SQ$=DIR$ THEN 5500:REM TELL
     USER FILENAME IS ALREADY THERE!
5050 GOTO 5020:REM KEEP TRYING TILL END
     OF FILE ON DIRECTORY
5500 PRINT "NAME IS ALREADY THERE,
     CHOOSE ANOTHER!":GOTO 3060
6000 PRINT "NAME YOU CHOSE IS OK"
6050 TRAP 3200
6100 GOTO 3110
```

If you RUN this program, you will see that if you choose a name such as:

```
FILE.EXT
```

it will search the directory of disk #1 to see if it is already there. If the name you choose is not there, it says that this name is OK to use for renaming an existing file.

(As a side note, this program can be used on its own, just to question the computer "does this disk contain a file by this name?")

To combine both programs, type:

```
ENTER "D.RENAME.LST" RETURN
20 RETURN
30 RETURN
3105 IN$ = NEUNAME$:GOTO 4000
```

then

```
SAVE "D:RENAMEIT"
```

As a test, type:

```
NEW
```

then

```
10 GR.0
20 PRINT "I WAS RENAMED!"
```

then

```
SAVE "D,BADNAME"
```

then

```
RUN "D:RENAMEIT"
```

Answer the questions:

```
1
BADNAME
GOODNAME
```

then

```
RUN "D:GOODNAME"
```

to see the results.

Here is a completed listing of the combined program so that you can check your work if there were any problems.

```
1 REM STAND-ALONE RENAME PROGRAM
2 REM PATCH FOR CHECKING BEFORE
  RENAMING A FILE IF NEUNAME ALREADY
  THERE!
3 REM 2 ROUTINES - ONE TO SQUEEZE AN
  INPUT STRING
4 REM OTHER TO LOOP THRU THE DIRECTORY
  TO SEE IF THAT NAME IS THERE
5 GRAPHICS 0:PRINT "SPECIFY A FILENAME
  STRING"
10 DIM IN$(20),FILE$(8),EXT$(3),
   DIR$(20),SQ$(20)
3005 GRAPHICS 0:REM CLEAR SCREEN
3010 PRINT "RENAME PROGRAM":PRINT
     :PRINT
3015 DIM Z$(1),OLDNAME$(12),
     NEUNAME$(12),RNAME$(40)
3020 PRINT "SPECIFY DRIVE NUMBER AND
     FILENAME"
3030 POSITION 2,6
3040 PRINT "DRIVE NUMBER (1 OR 2)"
3050 INPUT DRV
```

```
3055 IF DRV>2 OR DRV<1 THEN PRINT
     CHR$(253):GOTO 3030
3060 POSITION 2,8:PRINT "OLDNAME:"
3062 RNAME$="D"
3064 RNAME$(2)=CHR$(ASC("0")+DRV)
3066 RNAME$(3)=":"
3068 REM FORM THE EQUIV OF D1: OR D2:
3070 INPUT OLDNAME$
3075 IF LEN(OLDNAME$)=0 THEN PRINT
     CHR$(253):GOTO 3060
3078 RNAME$(4)=OLDNAME$
3080 POSITION 2,10:PRINT "NEWNAME:"
3090 INPUT NEUNAME$
3100 IF LEN(NEUNAME$)=0 THEN PRINT
     CHR$(253):GOTO 3060
3105 IN$=NEUNAME$:GOTO 4000
3110 ND=LEN(RNAME$)+1:REM SAVE SOME
     WRITING LATER
3120 RNAME$(ND)=",":RNAME$(ND+1)=
     NEUNAME$
3125 REM FINISHED, FORMING THE
     INSTRUCTION STRING, DO IT
3130 XIO 32,#2,0,0,RNAME$
3140 END
3200 PRINT "ERROR NUMBER ";PEEK(195)
4000 REM REFORMAT USER STRING TO
     EXACTLY MATCH THE DIRECTORY ENTRY.
4010 SQ$="           ":REM 11 BLANKS
4020 FOR I=1 TO LEN(IN$)
4030 IF IN$(I,I)="," OR IN$(I,I)=" "
     THEN GOTO 4060
4031 REM FORM THE FILENAME STRING PART
     , TERMINATE IF BLANK OR PERIOD
     FOUND
4035 IF I>8 THEN GOTO 4060:REM MAX 8
     CHARACTERS TO FILENAME.
4040 FILE$(I,I)=IN$(I,I)
4050 NEXT I
```

```
4060 REM NOW SEE IF THERE IS AN
     EXTENSION PART (SOMETHING AFTER A
     PERIOD)
4070 FOR I=1 TO LEN(IN$)
4080 IF IN$(I,I)="." THEN 4200
4090 NEXT I
4100 EXT$="   ":GOTO 4300:REM IF NO
     PERIOD, THEN NO EXTENSION ON NAME
4200 IF (I+1)>LEN(IN$) THEN GOTO 4300
4210 EXT$=IN$(I+1):REM LAST PART OF
     USER NAME MUST BE EXTENSION PART
4300 SQ$(3)=FILE$:REM FORM FIRST PART
     OF NAME
4310 SQ$(11)=EXT$:REM ADD EXTENSION
4311 REM NOW USER NAME MATCHES
     DIRECTORY ENTRY FORM, SO CAN SEE
     IF IT IS ALREADY THERE!
5000 OPEN #2,6,0,"D:*.*"
5010 TRAP 6000:REM TRAP END OF FILE
5020 INPUT #2,DIR$
5021 REM SHORTEN DIR$ TO 11 CHARACTERS
5030 DIR$(13)=DIR$(13,13)
5031 REM SHORTEN DIR$ TO 13 CHARACTERS
5040 IF SQ$=DIR$ THEN 5500:REM TELL
     USER FILENAME IS ALREADY THERE!
5050 GOTO 5020:REM KEEP TRYING TILL END
     OF FILE ON DIRECTORY
5500 PRINT "NAME IS ALREADY THERE,
     CHOOSE ANOTHER!":GOTO 3060
6000 PRINT "NAME YOU CHOSE IS OK"
6050 TRAP 3200
6100 GOTO 3110
```

NOTE AND POINT

There are two other functions which are associated with the disk system. These are named NOTE and POINT. They are associated with what are called *random access files*.

Only sequential access files have been discussed thus far in this book. A *sequential access file* is one in which the data is organized sequentially, one data item after the other. This is much like a film strip, where it is not possible to view any single frame until all preceding frames have been pulled through the slide viewer ahead of the one you wish to see.

Instead of a sequential organization, a *random access file* lets you go directly to any one data record without ever making a specific effort to bypass all the rest. This may be compared to an LP record, where all of the tracks on the record are visible. Once you have determined which "cut" you want to hear, you can move the needle directly to that cut without moving it through the others one at a time.

The examples just shown are very much the same as the data access used in the ATARI system.

Imagine all of the data on the cassette tape as a connected set of data strings attached to each other end to end. It is literally impossible to read these items in a random access mode, since each must pass across the playback head one record at a time before the next one can be reached. Therefore, accessing data records at random might mean reading many of them, then rewinding to get ready for the next read, and so forth. This would be very time consuming.

The ATARI disk system, on the other hand, has a moving head much like the needle on a record player. It can be moved directly to a particular spot on the disk and may be told to begin reading a particular item directly.

When a file is written, it is often written initially in a sequential manner. In other words, data items or groups of items form "records" within the disk file. In order to examine an individual record, you would like to have a choice of whether to read each record to see if it is the one you really want, or to have some way to speed up the data access by the random access method, instead. In the second case, you could go directly to the item in which you are interested.

ATARI DOS, as it writes the disk files, keeps a set of pointers internally which keeps track of where the next data item should be placed within the file. This pointer is accessible to the ATARI BASIC user with the NOTE command. Essentially, the NOTE com-

mand allows you to "make a NOTE of where the data item was placed." A complete set of NOTES about a file could serve as an index to the file. By looking up a data item in this index, one might POINT directly to the data item which is desired and therefore retrieve the data directly.

The NOTE command allows you to save the current location data pointers, and the POINT command takes whatever values you provide and replaces them in the current pointer, just the reverse of the NOTE command.

Using this kind of operation is one of the ways in which the programs called *Data Base Managers* can speed up their data access.

Take, for example, a file consisting of 300 lines of 120 characters each. If you would ask ATARI BASIC to read line 280, and you used the following approach, it would really take a long time:

```
4000 DIM A$(120)
4002 M = 280
4005 OPEN#1,4,0,"D:RAW.DTA"
4010 FOR N = 1 TO M-1
4020 INPUT#1;RECNBR
4025 PRINT RECNBR
4030 NEXT N
4040 PRINT "FINALLY FOUND RECORD # ";M
4050 INPUT#1;RECNBR;A$
4051 REM START THE STATEMENT WHICH WILL
     PROCESS THAT RECORD,
4052 REM REST OF LOOP WAS A SEQUENTIAL
     SEARCH
4053 REM UNTIL IT COUNTED OFF ENOUGH
     RECORDS,
4060 CLOSE #1
4070 IF M = 280 THEN M = 200:GOTO 4005
4080 END
4090 REM SEARCHES FOR ONLY TWO RECORDS,
     IN A SEQUENTIAL MANNER,
```

You will notice that even the do-nothing loop of

```
FOR C = 1 TO 300:NEXT C
```

takes some time to execute. Therefore, something that includes disk access on top of this must really take some time. If you had some way to go directly to the record number you wanted to get, though, the speed improvement would be simply dramatic. Here is how it can be done.

The plan of action here will be to create another file to go along with the data file, as it is created. Then, any time fast access to individual parts of the file is needed, this companion index file can be used to provide the data needed by DOS to get directly to the desired record.

To prevent a lot of retyping, you may use a program which was developed for Chapter 3 (MAKERAW.LST) as a basis for this demonstration. Or, you may simply use the parts here as a stand-alone demonstration of the power of the NOTE and POINT commands. The line numbers have been arranged to be compatible with either approach.

If you want to use this program separately, type NEW, then type in all of the lines shown in the next listing.

If you want to make it into the more useful program, then simply add lines 32, 205, and 215 to your MAKERAW.LST program from Chapter 3. This will build an index file, which you can use to rapidly go to any one of the data records in your file using a technique similar to that in the data readback program shown later in this section.

You can add these lines to a version of MAKERAW.LST which you have LISTed to the disk. If you have the program and have SAVEd it instead, then LOAD the program *before* you add the program lines mentioned previously. If it was LISTed to the disk, you may ENTER it before or after you type these other lines.

Here is the stand-alone program. The lines which can be added to MAKERAW.LST are shown first, for your convenience:

```
32  OPEN#3,8,0,"D:INDEXFIL.DTA"
205 NOTE#2 SS,BB
215 PRINT#3;SS;","BB
```

The following lines, if used exclusively with lines 32, 205, and 215, make this a 10-line program which can demonstrate the difference between sequential and random access:

```
30 OPEN#2,8,0,"D:RAW.DTA"
40 FOR N = 1 TO 300
50 RECNO = N
55 PRINT N
60 PRINT#2;RECNO;", IN THIS SEQ FILE,
   THIS IS RECORD # ";RECNO
220 NEXT N
250 CLOSE#1:CLOSE#2:CLOSE#3:END
```

Once this program has been RUN (assuming you have done it as a stand-alone version), you will have created a file called RAW.DTA which you can use for the search demonstrations. First, type in and RUN the program shown above.

Next, if you have not previously entered the sequential search program sample shown earlier (lines 4000–4090 on page 88), add it now to the RAW.DTA program you have just RUN. You can then RUN the sequential search sample by typing GOTO 4000. Do this now and notice how long it takes to RUN.

Now type NEW, then type in the following search version, instead. RUN it and see how long it takes to find and retrieve records for comparison. This version randomly retrieves 10 records, whereas the original version sequentially searches for only two:

```
5 REM FASTSRCH.DMO
10 REM NEW VERSION OF RECORD SEARCH
15 REM USES THE INDEX FILE
20 DIM SS(300),DD(300),A$(120):REM
   ALLOW FOR 300 RECORDS
30 OPEN#1,4,0,"D:INDEXFIL.DTA"
40 OPEN#2,4,0,"D:RAW.DTA"
50 TRAP 100:REM ASSUME ONLY END OF FILE
   ERROR
55 GR.0:PRINT:PRINT "READING INDEX
   FILE"
60 FOR N = 1 TO 300
70 INPUT#1,S,D
80 SS(N) = S:DD(N) = D:NEXT N
90 FOR PASS = 1 TO 10
```

```
95 REM READ 10 WIDELY SEPARATED RECORDS
   10 TIMES EACH
100 FOR M = 1 TO 10
110 READ REC
120 POINT#2,SS(REC),DD(REC)
130 INPUT#2,RECNO,A$
140 PRINT A$
150 NEXT M
160 RESTORE
170 NEXT PASS
210 CLOSE#1:CLOSE#2:END
500 DATA 279,199,3,120,45,100,295,
    22,187,6
```

As you can see if you tried this, there is a wide difference in the speed. It may have taken some time to read in the index file in the first place. But, once that data is in place, the disk system can be told exactly where each file is located.

POSSIBLE USE FOR AN INDEXED DATA FILE

You may have seen various programs on the market called *Data Base Managers*. These programs simply provide some easy way to save and recall data which the user has typed in or entered some other way.

Now that you know how quickly an indexed data segment can be found, let's try to imagine how this can be applied to the saving and finding of data.

A complete program will not be generated here; however, you will see various pieces that you might find helpful along the way to developing your own program. Notice that this is not necessarily the most efficient way of doing things, but it does demonstrate data retrieval.

To begin this project, you will have to decide what kind of information you want to save and retrieve. ATARI BASIC allows records to be up to 255 characters long, so you should choose something less than this as your maximum-data-per-record. If you discover that you need more space than this for any single

"set" of information, you should reserve more than one record for the data for each set.

Let's look at a possible application. How about a student list, to be maintained by a school? Each record may contain some of the following kinds of data:

Student Last Name: 16 characters max.
Student First Name: 12 characters
Address: 30 characters max.
Phone Number: 10 numbers (area code-3, number-7)
Student ID: 10 characters
Parent's Name: 30 characters

and so forth.

By writing a list of all of the information you will need to work with, you will be able to plan how much space will be needed to save it.

The most important thing to consider, however, is that there must be some way to retrieve the data quickly and properly. Here is one way this might be done. Look again at the list of the data items. If they are somehow all composed of characters, you can imagine taking each one of them and putting it together with others into one long string, as:

$\leftarrow$ last (16) $\rightarrow$ $\leftarrow$ first (12) $\rightarrow$ $\leftarrow$ addr (30) $\rightarrow$ etc.

After the data has been placed into a long string, that string can be written to the disk as a single record. This is just like the sentence which was used as the record in the example just shown.

There are many different ways that the data could be installed into the string. This section will first deal with the most obvious way of storing the data, then will look at other ways which might save some space on the disk and some time as well.

Notice in the list of items which you might want to store, that each item shows a maximum number of characters which will be in a particular field. (A field is one of the possible entries in the string being built.)

If you enter less than the maximum number of characters, something must be done to the output file to assure that the

computer will be able to recognize the address, the file ID, or any other field. To allow the computer to separate the data later, you must place the data in the precise place each time.

The following diagram illustrates this:

User enters:
Last name—Johnston
First name—Dick
Address—unknown (so user just entered a **RETURN** key)
Other fields (unknown), **RETURN**

When trying to place this data into an output record, you will want to put it in the correct order and correct position so that the computer can read it later.

First, let's assume that before each data record is entered, the output record is started, using a full line of blank spaces, and assume that the maximum size of an output record will be 100 characters:

Again, please note that a complete program is NOT being developed here, but you will be able to use these pieces to develop your own data tracker.

```
10 DIM OUTREC$(108),BLANK$(9)
11 BLANK$ = "           ":REM 9 BLANKS
3000 FOR N = 1 TO 99 STEP 9:REM 108
     BLANKS
3010 OUTREC$(N) = BLANK$:NEXT N
3020 REM FILL OUTPUT STRING WITH BLANKS
```

Now, assuming that the data was entered as mentioned, here is one way it can be put into that output string:

```
29000 DATA 16,12,30,10,10,30
```

Line 29000 is a data string representing the number of characters in each of the data fields (last name, first name), limited to only the data fields mentioned in the student list suggestion previously.

```
12 DIM START(6),LONG(6),CHLAST(6)
```

saves space for an array of numbers, shown in the start position of each of the output characters. The array also shows the posi-

tions *from which* the input data will be taken when the records are read back again. Line 12 holds the starting character, length of the string, and ending character.

```
15 CHPOS = 1:START ON FIRST CHARACTER
20 FOR N = 1 TO 6
25 START(N) = CHPOS:REM START POSITION
   FOR FIELD
26 READ X:REM GET FIELD SIZE
27 LONG(N) = X:REM SAVE IN THE LENGTH
   ARRAY
28 CHPOS = N+X:REM NEXT POSITION IS
   START+LENGTH
29 CHLAST(N) = CHPOS-1:REM LAST
   CHARACTER AT ONE LESS THAN FIRST OF
   NEXT FIELD
30 NEXT N:REM SET UP ALL POINTERS
```

With this kind of loop, you will have taken the original data list, showing the number of characters in each kind of data field, and converted it to a set of pointers, stored away conveniently in an array. In the example you are working with currently, it converts the numbers:

16 (first field)
12 (second field)
30 (third field)

and so on, into a list of:

START(1) = 1, LONG(1) = 16, CHLAST(1) = 16
START(2) = 17, LONG(2) = 12, CHLAST(2) = 28
START(3) = 29, LONG(3) = 30, CHLAST(3) = 58

and so on.

Why is this needed? Well, if you want to put data away in a long string, you will need to know for sure where in that string a particular piece of data starts, and where it ends. This is the only way you can be sure that you are getting back what you put there in the first place.

Let's say that you want to read the address part of the record,

which is now part of a string called INREC$. Assume that both INREC$ and OUTREC$ are the same length. Assume, also, that you have always been using OUTREC$ to build up a string for output as a record, then use INREC$ as the place into which OUTREC$ is read later.

Here, then, is a possible definition for INREC$:

```
5 DIM INREC$(108)
```

Now, to read the address portion of the record, you will want to be sure it can always be found in the same place in the incoming data record. This allows the computer to recognize it as an address.

To read that address, you would want to say:

```
6000 INPUT#1,INREC$

6400 ADDRESS$ = INREC$(29,58)
```

Because you have defined that the first 28 characters of the record take up the last and first name, the address here is taking up characters 29 through 58. (This example assumes that you have used a DIM statement to reserve 30 spaces for ADDRESS$).

But, what if you really wanted to generalize the program, and instead of ADDRESS$, you just wanted to call whatever you were reading as FIELD$?

Example:

```
6400 FIELD$ = INREC$(29,58)
```

This does not do too much as far as generalizing things. A more general-purpose routine would allow many different kinds of fields to be processed, using the same series of statements. The preceding example is only good for the address field, since it points only to that part of the incoming record!

Here is a more general-purpose statement, based on the array of START, CHLAST. It lets us define a set of processing statements just once. Then if you want to define other kinds of data handler programs, you will only have to put in a new DATA state-

ment, and change the number of fields and the size of the output record. Many of the other routines can remain the same since they will use the calculated data positions rather than ones which might have been *hard-encoded* (in other words, embedded throughout the program).

Better Example:

```
6400 FIELD$ = INREC$(START(FIELDNUM),
     CHLAST(FIELDNUM))
```

This line can now be used with any number of different fields, as long as the DIM statement for FIELD$ is at least as large as the largest field which it will expect to read.

Assuming that the variable FIELDNUM contains a value of 3, then the results of the evaluation of START(3) and CHLAST(3) are 29 and 58, respectively. This gives the same result and accesses the address section of the input string.

Now that you have seen that retrieving the data when it is input again requires that the user be sure where it is in the input field, let's look at how to assure it will go to the correct field as the output file is being built.

Here is a possible sequence which assumes that you have already grabbed the data from the user. This is written as a sub-routine which assumes that the variable INNUM contains the number of the input variable, and that INDATA$ contains the data itself.

Example:

```
12000 IF LEN(INDATA$) = 0 THEN RETURN
12001 REM DON'T TRY TO PUT AWAY
      "NOTHING"
12002 REM JUST LEAVE THE BLANKS IN THE
      OUTREC$ IN THAT POSITION
12010 IF LEN(INDATA$) <= LONG(INNUM)
      THEN 12030
12011 REM SEE IF INPUT STRING WILL FIT
      INTO THE SPACE PROVIDED
12015 CUT = LONG(INNUM)
```

```
12020  INDATA$(CUT,CUT) =
       INDATA$(CUT,CUT)
12021  REM IF NC CUT THE INPUT STRING
       DOWN TO THE MAX SIZE
12030  OUTREC$(START(INNUM)) =
       INDATA$:RETURN
12031  REM PUT INCOMING SINGLE DATA ITEM
       INTO OUTPUT STRING
12032  REM IN THE RIGHT POSITION
12033  REM RETURNS CONTROL TO THE
       CALLING ROUTINE.
```

Notice that CUT was only applied to any input data string which was *greater* in length than can fit into the field space which was saved for that field.

Why? If you use the string INDATA$ each time you call this subroutine, then the whole area reserved for INDATA$ will contain pieces of the previous string as well as the current contents, as the following example shows:

first time—INDATA$ = 114 COMPUTER ROAD
next time—INDATA$ = 1024 PERSONALITY CIRCLE
this time—INDATA$ = P.O. Box 62

where, the last time, the LEN(INDATA$) = 12. Since this is much less than the total size of the field (30) which is to hold the data, you must NOT stretch the size of the incoming data out, which means you should skip line 12020. This is because the area which has been reserved for INDATA$, unless *you* have *specifically cleared it to blanks* before you put the data there, will still contain leftovers (garbage) from the prior times that this string area has been used.

It will, from the example, actually contain:

P.O. BOX 62 ROADCIRCLE (maybe more characters)

Therefore, if you do execute the CUT line, it will extend the size of the input line, rather than to cut it down! This results in the garbage, which you don't want to put away.

For any data item which is shorter than the space reserved for

it, the blanks which you have placed into OUTREC$ will stay put. They only get written over when part of a line you wish to save replaces them.

Notice also that this technique *left-justifies* the input data. That is, it puts it with its first (leftmost) character as the leftmost character in the data field. This is very important when you are trying to compare two different pieces of string data to see if they are equal or different. This will be shown in the next chapter, "An Introduction to Sorting."

Thus far, this chapter has covered ways of putting away data strings, building an index file which shows where each of the strings starts, and retrieving the data using the index file. As a final note to this chapter, you might like to see some ways you can present your data requests to the user.

Note that the techniques and sample programs for this purpose have already been covered in the *ATARI BASIC Tutorial,* in the chapter titled "Menu Please." However, here I'll try to suggest a couple of the ways you might go about asking the user for the data.

One of the things you will want to do is show the user how many characters will be accepted into a particular space. This is used if it is your input processing routine which will limit the user's ability to enter more than you are asking for. As an example, you might produce the following line, either as part of a menu, or as part of a single data entry line, written each time this item is to be input:

```
LAST NAME:
?----------------
```

This illustration assumes that you are presenting the data question on one line, and accepting the answer on the next line. The hyphens in the second line tell the user what is the maximum number of characters which will be accepted as a reply. The "?" is just a representation to show you where you would POSITION your cursor in the line to show the user that this is where the input data must fit.

There are two reasons that a programmer might want to put the question (LAST NAME:) and the response line (---) on separate

lines. One reason is that the programmer might want to use the capabilities of the ATARI Screen Editor for the input. This does give control of the machine to the user until the **RETURN** key is pressed, but it also builds in the capability to use the backspace and tab functions in the data entry.

The second reason is related to the first. If you do use the Screen Editor, and put the question and the response line on one common line, then when the user hits return, he will enter as a string *everything* on that logical line, not just the response. You will then have to get rid of not only the extra hyphens, but also the question part, before you have the real string which represents the data item you want to save.

Example string returned, if question formed as:

```
LAST NAME: ?---------------
```

by the set of statements:

```
290  POSITION 2,19:PRINT:PRINT:
     PRINT:PRINT
295  POSITION 2,19
300  PRINT "LAST NAME:  --------------";
310  POSITION 13,19:INPUT INDATA$
```

If the user inputs JOHNSTON, then INDATA$ will contain:

```
LAST NAME:  JOHNSTON--------
```

and you will have to strip off the first 10 characters and the last 8, more or less . . . You will have to search and compare to see which of the letters is NOT a hyphen and keep all that are acceptable.

A better way is to really keep control over how the user enters data. In the preceding examples, the user could enter all kinds of control characters (cursor moves), or graphics characters, and so on, before pressing **RETURN**. The cursor might be on some line of real junk somewhere on the screen, and your program, using the Screen Editor for its input (using the INPUT statement directly) would accept whatever string it was thus presented.

You can keep control by using the GET function for each of the characters you input from the keyboard, such as:

```
10  OPEN#1,4,0,"K:"
11  REM OPEN AN IOCB JUST FOR THE
    KEYBOARD
20  GET#1,KEYM
21  REM READ A SINGLE KEYSTROKE,
```

An example program showing how you can keep control of the keyboard during user input is shown on pages 173–175 of the *ATARI BASIC Tutorial*. Basically, by reading each keystroke individually and interpreting its meaning, you can count each legitimate key, adding to the string. Or you can interpret the **DELETE/BACK S** key as a deletion/backspace within the output string, and thus never have to lose control or throw away any of the string after the user finally hits **RETURN**. This program is essentially adaptable to a menu kind of a data gathering approach, because of the control it offers.

One final remark about this kind of data gathering. If you do choose to produce a menu for the data input, you will be able to use the same menu for later presentation of the data to the user, since he or she will already be familiar with the form which was used to input the data in the first place. When the user looks at the form, searching for a particular item, he or she will already know in which position on the screen this information should appear.

You will see more about searching for items in the next chapter.

REVIEW OF CHAPTER 4

1. All of the ATARI DOS commands work either with a string variable in the command or with the string spelled out.
   As an example, you may say:

   ```
   OPEN#2,4,0,"D:SOMENAME,BAS"
   ```

   or you can say

   ```
   OPEN#2,4,0,M$
   ```

   where

   ```
   M$="D:SOMENAME,BAS"
   ```

This allows you to write a general-purpose program, and build the strings as needed to work correctly with the disk system.

2. To delete a file from ATARI BASIC, use the command:

```
XIO 33,#2,0,0,FILENAME$
```

where #2 can be any of the IOCB numbers from 1 to 5, and FILENAME$ is the string description of the file name you wish to delete.

3. To rename a file from ATARI BASIC, use the command:

```
XIO 32,#2,0,0,NAMESET$
```

where

```
NAMESET$ = "D1:OLDNAME$,NEUNAME$"
```

Remember: Keywords can't be used except in quotes as part of a PRINT statement.

4. To LOCK a file (prevent DOS from erasing it) or UNLOCK a file (allow DOS to erase it), use the XIO commands 35 and 36, respectively. They will have the form shown in item 2 above.

5. Using the NOTE command, you can make *note* of where, in a file, the current data will be placed. Then later, using the NOTE information, you can use POINT to move the file *pointer* directly to this position, saving a lot of time which would be required if only sequential searches were possible.

6. You can use the DOS functions to build relatively short programs which can help you keep track of large amounts of data, and present the data items in an easily understood manner. This is often associated with what is called a "Data Base Manager" program.

CHAPTER 5

An Introduction to Sorting

This chapter will show you how some of the data handling techniques you have learned here and in the first book can be made more useful.

Chapter 4 introduced the idea of creating a data file out of pieces of information which were obtained from the user, and showed how an index file would be built using the NOTE command, and then used for fast data retrieval with the POINT command. This section will suggest ways in which large amounts of data can be effectively searched or sorted to make that data more useful to you.

To get to that point, the technique of sorting must first be discussed. The sorting technique which will be used here is not the most efficient one, but it is rather easy to understand. The goal of the sorting will be to produce an index file which will allow you to access rapidly the data in a much larger file. Therefore, I felt that the speed of the sort routine itself was secondary to the goal of speeding up the data access to a larger file.

You may find other books which will explain and implement some of the more efficient sorting methods. Hopefully, you will gather enough information from these ATARI BASIC tutorials to be able to understand and use these other, more sophisticated, programming techniques for your future programs.

When considering sorting, it is usually best to start with a small example. Then, the small example can be expanded to perform a more useful function without increasing its complexity.

An example of something which you may wish to put into numerical order might be the following set of numbers:

19   3   24   6   12   9   1   15

You will see two different kinds of sort functions performed here. One of them, called the *bubble-sort,* can be used when there is very little memory which can be dedicated to the sorting process. This kind of sort lets the smallest (or largest, if you wish) item rise (similar to an air bubble in water) to the top of those "heavier" (larger) items, with each next larger item below the smaller one.

The second type of sort takes items one at a time out of a group and builds a new list of all the items. If you time both, it is possible that one may be faster than the other. It depends on the initial order of the data, as you will see next.

THE BUBBLE SORT

Again, look at the list of numbers proposed earlier:

19   3   24   6   12   9   1   15

If they are to be put into ascending numerical order, one way is to start at the head of the list and compare the first one to the second one. If the first is larger than the second one, then you would know that they are in the wrong position and should be swapped.

Doing this the very first time results in the new list of numbers (from here on I'll call it an ARRAY), in this form:

<u>3   19</u>   24   6   12   9   1   15

The underlined sections of the preceding example and the next will tell you which items were compared for that particular array examination. Each shows the *result* of the comparison, so it will be obvious whether or not a swap has taken place since the preceding line, which shows the "old" array data.

Now, if you continue on down the array of numbers, making the same comparison for each sequential pair, you will form the following set of entries as the count proceeds to the end of the array:

```
3    19    24    6     12    9     1     15
3    19    24    6     12    9     1     15
3    19    6     24    12    9     1     15
3    19    6     12    24    9     1     15
3    19    6     12    9     24    1     15
3    19    6     12    9     1     24    15
3    19    6     12    9     1     15    24
```

As you can see, when all adjacent pairs of data items have been compared one time, the highest numbered item has bubbled itself to the back of the list. At least one of the items is now in the correct position.

However, before ending a sort, you must be sure that ALL of the data items are where they belong. Let's see how this might be done.

First, since the highest numbered item has already gone to the bottom of the list, you will not have to look at it or handle it any more. This means that there will be one item fewer to look at than there was when you started. Also, it means that the second "pass" through the array will be one step faster.

Let's look at a program piece which will perform that same comparison and data swap which was shown previously:

```
NEW  RETURN
10 DIM ARRAY(8)
20 ARSIZE = 8:REM DEFINE ARRAY SIZE
30 FOR N = 1 TO ARSIZE
40 READ Q:ARRAY(N) = Q:NEXT N
41 REM READ THE DATA FROM THE DATA
   STATEMENT
30000 DATA 19,3,24,6,12,9,1,15
50 FOR N = ARSIZE TO 2 STEP -1
70 FOR P = 1 TO N-1
80 IF ARRAY (P+1) > ARRAY(P) THEN 120
81 REM COMPARE TWO SEQUENTIAL NUMBERS
   IN THE ARRAY
```

```
82 REM IF GREATER NUMBER IS IN HIGHER
   SPOT, EXIT
90 TEMP = ARRAY(P+1):REM USE TEMP AS A
   PLACE TO
91 REM HOLD ONE VALUE SO CAN DO SWAP OK
100 ARRAY(P+1) = ARRAY(P):REM FIRST
    PART OF SWAP
110 ARRAY(P) = TEMP:REM COMPLETE THE
    SWAP
111 REM ONLY USED ONE EXTRA SPACE
    (TEMP)
120 NEXT P:REM DO THE NEXT SEQUENTIAL
    PAIR
140 NEXT N:REM DO THE NEXT PASS FROM
    THE START
```

Because there is an inner loop which gets smaller each pass, it ignores the largest item which gets bubbled forward to the current end each pass. When it has made all of its passes, it will stop, with all items in the proper order. Here is the end of the example, which prints the values in the correct order:

```
150 GOSUB 160:END
160 FOR Z = 1 TO ARSIZE
170 PRINT ARRAY (Z);"   ";
180 NEXT Z
190 PRINT:PRINT
200 RETURN
```

NOTE: Z is used to show that you can have multiple variables within a program, but each variable must be part of its own unique loop.

Try the program if you wish. It will do all of the things you saw in the "done-by-hand" set of listings which were shown earlier.

In fact, if you wish to see a progression showing how the program did the operations, you may change line 120 to read:

```
120 GOSUB 160:NEXT P:REM PRINT INTER-
    STEP TOO
```

It will list the intermediate steps, as well as the final printout.

Now imagine a situation where you have 998 different numbers in the array, and you add two numbers to the back of the array. Also, let's say that when you added these two numbers to an already numerically sorted array that they are higher than any number previously in the array. Therefore, when you add them to the array, at worst, it will require one pass to get them into the proper order.

Well, in its present form, the program simply does not know any better, and will assume that it must make 999 passes through the array. That will take a lot of time when only one pass (or at most 2) will be enough.

Therefore, you will have to make a small modification to the program to assure that this waste of time does not happen.

Add the following lines to the program:

```
45 YES = 1:NO = 0
60 ANYSWAP = NO
85 ANYSWAP = YES
130 IF ANYSWAP = NO THEN GOTO 150
```

Line 45 defines the words YES and NO, so that the program can internally document itself. Notice that this gives lines 60, 85, and 130 some real meaning.

Line 60 is at the top of the full-pass loop (starts where the compare of a complete array begins). This says ANYSWAP = NO. This indicates that, at the top of the loop, no data items have yet been swapped during this pass.

Line 85 says ANYSWAP = YES. Each time it goes through this set of lines, it will repeat YES. It is an extra step, added to the swap, but it does give us a way to test at the end of a loop whether anything happened or not.

This way, when the program gets to line 130, if no swap takes place during the first pass at 999 number pair combinations, then it means that all of the numbers are already in order!

If there has been one or more swaps, then ANYSWAP will be YES, and it will go back and scan the list again and again, until it is really done with the reordering of the data.

This allows the bubble sort to exit when it knows it is finished, rather than complete the loops and simply waste time.

AN INSERTION SORT

The next form of sorting which will be shown here is a form of an *insertion sort*. This indicates that you will be trying to do a program which will take each data item and *insert* it into its proper place in the array.

Let's continue to use the same data set as was used for the bubble-sort:

3   19   24   6   12   9   1   15

An insertion sort consists of looking at each item, one at a time, and comparing its current position to that of all of the items previous to it in the array.

When you find a spot in which it will fit perfectly, with an item less than this one to its left, and one item greater than this one to its right, it will be in the right place and can be inserted.

Imagine a hand of playing cards. Fan the cards, such as the preceding set, out in front of you. Now start with card number 2 (in this example, the number 19) and compare it to all of the cards previous to it, one at a time. In this case, it is the number 3. Three is less than this number, and among the numbers being compared (3 and 19), it is currently in the correct sequence (ascending).

Now look at the third card (24). Comparing it to the others that came before it in the array, one at a time, it is greater than the last item (19), so it should not be moved either.

Now look at the fourth card (6 in this case). Examine those in the current sequence from the beginning. The new number is greater than 3, so it does not belong ahead of it. It is less than the next number (19). So, since we found a slot into which it can fit (between 3 and 19), this is where it must be moved.

Now the array reads:

3   6   19   24   12   9   1   15

The next item to be inserted is the fifth item (the 12). It too is checked against the array of previous items:

3   6   19   24
     12

It now shows where the insertion would occur in this sequence.

Each of the items in the array is "picked up" and *inserted* into the array. Once the final item has been inserted, the sort is done.

At first glance, this would appear to require fewer data moves than the bubble sort. However, if the memory space is somewhat tight (not much space available), you may need to sort the data where it sits, as in the bubble sort. The following example shows how this may require quite a few data moves anyway.

Let's look at the insert of the fourth item (the 6) into the array, just to see the sequence which is needed for this one item:

```
3    19    24    6
     ↑
     └ the 6 belongs here.
```

So, if we don't have much memory to work with, we can move the 6 into a temporary location

```
TEMP = ARRAY(4)
```

This makes the array look like this:

```
3    19    24    (empty slot)    12    9, etc,
```

We have identified that the 6 belongs in slot number 2, so now we have to move some data to make room for it.

```
3    19    empty    24    12    9, etc,
```

Move the 24 into the empty slot.

```
3    empty    19    24    12    9, etc,
```

Move the 19 into the slot vacated by the 24.

```
3    6    19    24, etc,
```

Move the 6 from TEMP into the slot where it belongs.

Essentially, as each item is being examined, if it is *not* in the slot where it belongs, then move it out of that slot, making some room in the array to use for shifting everything else around. This seems to make an empty slot which is continuously shifted down

the array until it is in the right spot to make room for the item
which belongs there.

Let's see how this translates into a BASIC program. Some of
the same program lines from the BUBBLESORT program will
be repeated here, to save you some typing. These lines are
identical:

```
10 DIM ARRAY(8)
20 ARSIZE = 8:REM DEFINE ARRAY SIZE
30 FOR N = 1 TO ARSIZE
40 READ Q:ARRAY(N) = Q:NEXT N
41 READ THE DATA FROM THE DATA
   STATEMENT
150 GOSUB 160:END
160 FOR Z = 1 TO ARSIZE
170 PRINT ARRAY (Z);"    ";
180 NEXT Z
190 PRINT:PRINT
200 RETURN
30000 DATA 19,3,24,6,12,9,1,15
```

And these lines comprise the INSERTION SORT:

```
50 FOR N = 2 TO ARSIZE STEP 1
70 FOR P = 1 TO N-1
80 IF ARRAY(N) > ARRAY(P) THEN 120
81 REM COMPARE CURRENT ITEM TO ALL
   ITEMS AHEAD OF
82 REM IT WITHIN THE CURRENT ARRAY. IF
   IT IS
83 REM GREATER THAN ALL OF THEM, IT IS
   ALREADY
84 REM IN THE RIGHT SPOT AND NOTHING
   MOVES
90 TEMP = ARRAY(N):REM USE TEMP AS A
   PLACE TO
91 REM   HOLD ONE VALUE SO CAN OPEN A
   SLOT FOR
92 REM   MOVING THE REST OF THE ARRAY.
```

```
100 FOR R = N TO P+1 STEP -1
105 ARRAY(R) = ARRAY(R-1):NEXT R
106 REM MOVE ALL DATA FORWARD ONE STEP,
107 REM STOP WHEN "EMPTY SLOT" WINDS UP
    IN
108 REM THE RIGHT PLACE.
110 ARRAY(P) = TEMP:REM AND INSERT THE
    DATA
112 GOSUB 160:REM PRINT INTERMEDIATE
    RESULTS.
115 GOTO 140:REM GO FOR NEXT ITEM
120 NEXT P:REM LOOK AT THE NEXT
    CHARACTER
140 NEXT N:REM DO THE NEXT ITEM
```

You may try either type of sort and see for yourself which one would be more efficient in your own application.

ALPHABETIC SORTING

The previous sorting examples will work equally well for alphabetic as for numeric sorting. All that must be done is to change the references to numbers into references to strings.

If you refer to the BUBBLESORT program, you can make the following changes and do *alphabetic sorts* instead. To convert the BUBBLESORT program so that it will handle strings instead of numbers, the first step is to change the name of the ARRAY into ARRAY$.

The second step is to make sure that the correct number of spaces is reserved in the string as the string must hold totally (number of characters per word times number of words). If you want to sort 16 character strings, then the example needs a string of 16 × 8 or 128 characters.

The next thing to be done is to provide additional arrays for the start and end characters of each word. (Since all of the words will be in a single long string, the program must have a way to know where one ends and another begins.) For this purpose you may use the number arrays START(8), and CHLAST(8) which were introduced in the last chapter.

The next listing is a rewrite of the BUBBLESORT program adapted for strings. The parts which are similar have been given the same line numbers as used in the BUBBLESORT, so that you can compare the new program to the old one.

The sections which are new are specifically made a separate section so that they may be explained individually.

```
5 DIM START(8),CHLAST(8):REM SAVE SPACE
  FOR POINTERS
6 PINTER = 1:REM AVOID KEYWORD POINT
  SPELLING
7 FOR MM = 1 TO 8
8 START(MM) = PINTER:CHLAST(MM) =
  PINTER+15
9 PINTER = PINTER+16:NEXT MM:REM SET
  POINTERS FOR STRING STORE & COMPARE
10 DIM ARRAY$(128),Q$(12),TEMP$(12)
20 ARSIZE = 8:REM DEFINE ARRAY$ SIZE
21 REM SAME AS NUMBER EXAMPLE BUT
   ADAPTED TO ARRAY AS A STRING
30 FOR N = 1 TO ARSIZE
35 ARRAY$(START(N)) = "             "
36 REM START WITH ALL BLANKS (16)
40 READ Q$:ARRAY$(START(N),CHLAST(N)) =
   Q$:NEXT N
41 REM READ THE DATA FROM THE DATA
   STATEMENT
42 REM LEFT-JUSTIFY THE DATA.
45 YES = 1:NO = 0
50 FOR N = ARSIZE TO 2 STEP -1
60 ANYSWAP = NO
70 FOR P = 1 TO N-1
80 IF ARRAY$(START(P+1),CHLAST(P+1)) >
   ARRAY$(START(P),CHLAST(P)) THEN 120
81 REM COMPARE TWO SEQUENTIAL STRINGS
   IN THE ARRAY$
82 REM IF GREATER STRING IS IN HIGHER
   SPOT, EXIT
```

```
85 ANYSWAP = YES
90 TEMP$ = ARRAY$(START(P+1),CHLAST
   (P+1)):REM USE TEMP$ AS A PLACE TO
91 REM    HOLD ONE VALUE SO CAN DO SWAP
   OK
100 ARRAY$(START(P+1),CHLAST(P+1)) =
    ARRAY$(START(P),CHLAST(P))
101 REM FIRST PART OF SWAP
110 ARRAY$(START(P),CHLAST(P)) = TEMP$
111 REM COMPLETE THE SWAP
120 NEXT P
130 IF ANYSWAP = NO THEN GOTO 150
140 NEXT N:REM DO THE NEXT PASS FROM
    THE START
150 GOSUB 160:END
160 FOR N = 1 TO ARSIZE
170 PRINT ARRAY$(START(N),CHLAST(N))
180 NEXT N
190 PRINT:PRINT
200 RETURN
30000 DATA VALUE19,VALUE3,
      VALUE24,VALUE6
30100 DATA VALUE12,VALUE9,VALUE1,
      VALUE15
```

The same number sequence which was used in all previous examples is shown here. If you RUN this program, you will get the following results:

```
VALUE1
VALUE12
VALUE15
VALUE19
VALUE24
VALUE3
VALUE6
VALUE9
```

These VALUES are not at all what you might have expected, are they? Well, take a closer look at these VALUES and you will

see that the program really did put them in order the way it was designed to do.

You see, when ATARI BASIC makes a comparison between two strings, it looks at *each* character in the strings, including the blank spaces. For example, the strings:

        VALUE24 and VALUE3

are actually composed of the following strings of ATASCII characters:

```
VALUE24 is:  V    A    L    U    E    2    4
            86   65   76   85   70   50   52   20   20   20
VALUE3 is:   V    A    L    U    E    3
            86   65   76   85   70   51   20   20   20   20
```

The values of 20 in the list represent the blank spaces which are stored as a part of the string.

ATARI BASIC treats each character like a number in a specific position when it is trying to compare if one string is larger than another. It begins with the leftmost character and keeps going to the right until it gets either to a point where the greater-than or less-than status can be assured, or to the end of the string, if an equal-to status is found. So, if you just look at the characters from the "E" onwards, you can see that

        VALUE3 is greater than VALUE24
        (70 51 20 > 70 50 20)

so it must belong later in the alphabetic listing.

You can fix the problem of putting the value figures in numerical order by specifying them with leading zeros. Notice that you will need to use enough zeros to "fill" in the smallest number possible, either with blanks or the leading zeros (either one or the other but not both) to assure that the rightmost digit of the number parts are at the same character position. Such as:

        VALUE00009
        VALUE12345

instead of VALUE9 and VALUE12345, since this second kind would not alphabetize in the way we want it to.

This is the reason you saw the caution note elsewhere in this book about aligning the data words if you wish to do a sort. You see, if you do the same kind of thing with purely alphabetic information, and do not align them (left-justify the data), you will have the same problem. An example is:

```
A  B  C
   X  Y  Z
|  ←——— common start position in two different strings (that
        is, X Y Z has a leading blank space).
```

If you had accepted these two strings from the user and wanted to alphabetize them, ATARI BASIC, literally, would find:

```
20 88 89 90        for  XYZ
              and
65 66 67 20        for  ABC
```

Since the value of 20 88 89 90 is less than the value of 65 66 67 20, ATARI BASIC would place the XYZ string ahead of ABC in an alphabetic listing.

This is not desired, and can be corrected if you *always keep the string beginnings correctly aligned.* You can imagine the difficulty in producing an index to a book from a list of titles. If an alphabetic sort program is used and the writer of the list doesn't keep the fields properly justified (or make other special provisions), how will anyone ever be able to use the index properly?

CHAPTER 6

Sorting: the Next Logical Step

A goal of this book is to teach you all you would need to know to build a simple, but useful data base manager program.

Looking back at the topics covered so far, you have seen:

- How to build and edit a sequential data file.
- How to read a sequential data file.
- How to build an index file to allow quick positioning of the file to a selected record.
- How to sort data in numerical/alphabetical order.

The next logical step is to give you one more tool which will let you combine the index file and the sort capability. This will let you retrieve your original data in various different ways.

For example, let's use the student listing data file, which was the subject of earlier examples. If you build a sequential file using this format, you may someday want to retrieve the data:

1. In alphabetical order by student last name.
2. In numerical order by zip code.
3. In numerical order, by first 3 digits of the phone number.
4. Any other form of data grouping specification.

When you are trying to retrieve data using only certain pieces of a record, those pieces of the record are called *keys*. Once the

115

keys have been separately identified, they can be used with the record pointers, to provide a quick way to re-order the data *access* method without changing the order of the data in the disk file itself.

In other words, once the sequential file is written, the data may be taken from anywhere within the file by using the index and key information. An example makes this clear:

Let's say that there are many records in the student list file, and assume that the file is to be displayed by the student name, in alphabetical order. You will want to make up an index file such as:

```
LASTNAME STARTS STARTB

JOHNSTON 141       3
SCHMIDT   93      18
FLORBUSH  93     121
ABRAMS   217       9
RENKO     52      84
```

In this case, the LASTNAMEs are exactly the names which come out of the first part of each of the student list records on the disk. The STARTS and STARTB columns stand for the starting sector and byte position on the disk for that record from which the name has been taken. Note that the *values* in STARTS and STARTB are of *no concern* to you, other than the fact that you *must associate each with the record itself*. Thus, if you want to quickly read that record, those values will be used with a POINT command on a file which has already been OPENed for a read.

We now come to a point where some time can be saved simply by considering carefully the way in which data comparisons and moves will be made.

For example, if you were going to compare 100 strings each with 100 letters, it would probably take quite a long time if each of the "swaps" were made using TEMP$ as shown previously, then moving the rest of the string pairs around.

There is an alternative to this, and this alternative is what will be shown for developing index files for fast data access in such

things as the student file. The alternative is to assign a WHERE
array for each one of the strings, as:

```
KEY-COMPONENT   WHERE

JOHNSTON          1
SCHMIDT           2
FLORBUSH          3
ABRAMS            4
RENKO             5
```

I have called it the WHERE array because initially it tells us
where, in a sequential list of the keys, the record is located. As
you will also soon see, it will tell us where the keys should be
found for each of the comparisons we will do.

In particular, when comparing strings, the WHERE array will be
examined to determine which of the string pieces within the KEY$
array will be compared. You will next see exactly how this will be
done.

Assume that you have already set up a long string called KEY$.
It must be long enough to hold all of the keys which you wish to
use for the sort. If, for example, you had 300 records, assume
that you want to sort the records based on the student's last
name.

First you must figure out how many characters of the last name
are the minimum you can get away with but still be assured that
a sort will result in a unique alphabetic list. Let's assume that this
minimum is 8 characters. (If you can get by on less, use less . . .
the more characters used, the more time the program must take
for comparisons.)

If you need 8 characters, then the KEY$ array must be at least
2400 characters long. In addition, you will need two other arrays:
one for STARTS and one for STARTB. Each of these arrays will
be 300 numbers long, and will be used to hold the data found in
the index file (the data from the NOTE command). For purposes
of memory planning, each of the 600 locations reserved for
STARTB(N) and STARTS(N) uses 6 memory locations, so these
arrays use 3600 locations in all.

Therefore, it will take at least 8000 memory locations just to hold the data which you want to sort. You should take this into account when you plan your program.

There is at least one more set of memory locations which will be needed to complete the setup for this key-sort technique. This is the WHERE array. It will need 300 number array locations (times 6 per number), or 1800 memory locations. Thus about 10,000 locations may be needed to allow a sort of 300 records, using a single 8-character key.

But enough of the planning angle. Most of you have 48K systems. Even a 16K system, such as the ATARI 400, will be able to run a 300-record sort and have some room to spare. Let's see exactly how the sort will occur.

Using the previous example, with simply the names which are already in the list, here is how they would be strung out end to end in a single string. (Coincidentally, the names are all 8 characters or less, so none will be shortened.)

```
JOHNSTONSCHMIDT   FLORBUSHABRAMS   RENKO
1         9        17     25       33      (contents of
                                            array START)
          8        16     24       32      40(contents
                                            of array
                                            CHLAST)
```

The array called START is, of course, the one you have seen before that always points to the starting character of a string embedded into a larger string array. CHLAST always points to the last character.

As you recall, any word number in the large array can be located by looking up the position of the start letter in the START array, and the last letter in CHLAST.

You may have wondered why there is not just one array, called START, used for both purposes. After all, CHLAST array contents are always just START(N)+WORDSIZE and this seems to be an easy item to calculate each time. Such a word access might be:

```
GET WORD 4 (ABRAMS),,,
ANSWER$ = KEY$ ( START (4),
    START(4)+WORDSIZE )
```

The reason that the separate CHLAST array is set up is that any interpretive BASIC is somewhat slow, and it is faster to simply look up CHLAST(4) than to calculate it as START(4) + WORDSIZE each and every time. Since there will be many such references, any amount of calculation time that may be saved will cut time out of the whole process.

Once you have the pointer arrays START and CHLAST set up, and the data read into the KEY$ array, you can begin to sort.

First, initialize the WHERE array, using a sequence like this one:

```
300 FOR N = 1 TO 300:WHERE(N) = N:NEXT N
```

Line 300 makes WHERE(1) = 1, WHERE(2) = 2 . . . WHERE(300) = 300.

Now here is the meat of the sorting technique. You will:

1. Take items sequentially, two at a time from the WHERE array. These items are position numbers.
2. Compare the KEY$ pieces ***AT THOSE TWO POSITION NUMBERS*** to each other.
3. If the KEY$ item at the higher of the "selection numbers" is greater, then do nothing. If the KEY$ item at the higher of the two selection numbers is lower, then swap the numbers ***IN THE WHERE ARRAY*** *(not in the KEY$ array)*!!!!
4. Treat the passes through these arrays-in-combination just like a bubble sort, continuing the passes until no swap takes place.

Before going on, two definitions are needed:

SELECTION NUMBER means the position within the WHERE array from which the pointer data is being taken.

POSITION NUMBERS are the actual contents of the WHERE array itself, which are the pointers to the records within the KEY$ array.

A complete alphabetization example will be shown later. Just the names seen so far have been used, so that these items can be clarified further.

Here, again, is the array. The left column shows the individually isolated names which are embedded in KEY$, along with a column which shows how they are isolated.

Table 6-1. Complete Alphabetization Example

P	How Name Found	Name	Current Content of WHERE(N)
1	KEY$(ST(1),CL(1))	JOHNSTON	1
2	KEY$(ST(2),CL(2))	SCHMIDT	2
3	KEY$(ST(3),CL(3))	FLORBUSH	3
4	KEY$(ST(4),CL(4))	ABRAMS	4
5	KEY$(ST(5),CL(5))	RENKO	5

In Table 6-1, I have abbreviated START to ST, and CHLAST to CL, just to make some room. In the program which shows the complete technique, they are abbreviated in this same way. (It could save you some typing.)

Now that you can see how the string data is accessed from the long string, only columns 1, 3, and 4 will be shown for the rest of the examples.

P, in Table 6-1, represents the normal sequence of positions which will be searched, from beginning to end in each pass, until it is known that the sort is completed.

In other words, when a pass of this sort is to take place, it begins with the item at P = 1 being compared to the item now at P = 2. Then the second step of a pass takes the item currently at P = 2 and compares it to the item at P = 3, and so on. There will, of course, be an ANYSWAP flag to tell us whether and when we are done. In this case, the "item" to which we are referring is the *value* at that location in the WHERE array, as you will see.

Table 6-2 shows the results of the sort after one complete pass through the source data. The numbers in the rightmost column represent the current ranking of the data keys after this first pass.

In the columns titled for first compare, second compare, and so on, the chart shows which KEY$ original entry numbers are being compared, and what is in those positions in the columns shows the results of the comparisons.

All four comparisons will be explained here, just to be sure that you will be able to follow what is happening.

For the first comparison, the contents of WHERE(1) and WHERE(2) are retrieved. Each is a pointer to one of the keys in

Table 6-2. Sort Results After One Complete Pass

P	Name	Current Value WHERE(N)	After First Compare	After Second Compare	After Third Compare	After Fourth Compare	Ranking for Next Pass
1	JOHNSTON	1	1				1
2	SCHMIDT	2	2	3			3
3	FLORBUSH	3		2	4		4
4	ABRAMS	4			2	5	5
5	RENKO	5				2	2

the KEY$. In this case, WHERE(1) points to JOHNSTON, WHERE(2) points to SCHMIDT.

SCHMIDT is greater than JOHNSTON. Since the pointer to SCHMIDT is at a higher "P-VALUE" (at WHERE(2)), than JOHN-STON (pointer at WHERE(1)), nothing has to be done. This is equivalent, in the bubble sort, to finding two items already in the right place.

For the second comparison, the contents of WHERE(2) and WHERE(3) are used. WHERE(2) now contains a 2, which points to SCHMIDT. WHERE(3) currently contains a 3, which points to FLORBUSH. Since SCHMIDT > FLORBUSH, swap the contents of WHERE(2) with the contents of WHERE(3), as seen in the table as the result.

The third compare takes the current contents of WHERE(3) which is now 2 (points to SCHMIDT), and the current contents of WHERE(4) which is 4 (points to ABRAMS). SCHMIDT > ABRAMS, so swap the contents of WHERE(3) and WHERE(4).

The fourth comparison uses items 4 and 5 of the WHERE array. The contents of WHERE(4) is 2 (from the above comparison and swap); the contents of WHERE(5) is 5, points to RENKO. SCHMIDT > RENKO, so swap the contents of WHERE(4) and WHERE(5).

Ranking of the data going into the next complete pass is now: 1 3 4 5 2

Notice that no string data has been moved at all, just numerical data. Let's take a quick look at this present alphabetic ranking to see how it is going:

JOHNSTON FLORBUSH ABRAMS RENKO SCHMIDT
    1   3   4   5   2

If you refer to the BUBBLESORT routine, you will see that after the first complete pass, the highest of the values will have been pushed to the end of the number group.

Well, that is exactly what has happened here. The name SCHMIDT has been pushed to the last position if you use the WHERE array as the index into the original data sequence.

This technique of indexed referencing will be shown again after the full sort has been completed.

Only two more full passes are required before the data will be fully in order. These will be written horizontally, as in the first sorting example, rather than vertically, to save room. Follow along with the chart just presented so that you can see what names are being compared, and why the results are as they are shown.

PASS 2
```
    start position      1       3       4       5       2
                        3       1       4       5       2
                        3       4       1       5       2
PASS 3
    start position      3       4       1       5       2
                        4       3       1       5       2
```

The lines below the numbers show the positions where both a compare and a swap HAVE taken place. All of the steps with the intermediate *compare/no swap* have been left out (example—compare table entry 2 to entry 5 SCHMIDT > RENKO . . . already in the right place in the table).

Now that the sort is complete, you will have built a new table called WHERE. And each item in the array becomes the pointer to the record number which you want to get from the file, as though the records themselves had been swapped.

So the final ordering of the WHERE array says that to retrieve the information for the first name (get info from pointer in WHERE(1) points to record number 4, which contains ABRAMS). All of the other record pointers are also in the correct order.

Now that the plans for the program have been shown, it is time

to do the program itself. At the end of the program will be a set of DATA statements which will simulate the sort. You may want to try this using your own data file later.

As usual, a lot of commentary will be inserted between the lines. Because of the comments, the REM statements will be minimized within the program itself.

```
5 DIM START(300),CHLAST(300)
```

Set up the arrays for the start and end characters.

```
10 DIM KEY$(2400),WHERE(300)
```

Set up the KEY string, and the WHERE string.

```
20 DIM STARTB(300),STARTS(300)
```

Set up the arrays for holding the values from the NOTE command when the file was first built.

```
30 DIM CUT$(8)
```

Set up a string to cut the input length down to the maximum number of characters expected by the sort.

```
100 OPEN#2,8,0,"D:DUMMY.DAT"
```

Set up a dummy file as an example.

```
110 OPEN#3,8,0,"D:DUMINDEX.DAT"
```

Open a dummy index file for the NOTE information about the dummy data file.

```
120 DIM DUM$(108)
```

Use a dummy string for holding the data itself.

```
130 FOR N = 1 TO 10
```

This program only concerned with 10 records.

```
140 NOTE#2,S,B
```

Find out where the data output file is now pointing. Make a note of it for the dummy index file.

```
150 PRINT#3;S;",";B
```

Put away the dummy index entry.

```
155 READ DUM$
160 PRINT#3;DUM$
```

Put away the dummy data entry also.

```
170 NEXT N
```

Do the next entry.

```
180 CLOSE#2:CLOSE#3
```

Close both files on completion.

```
20000 DATA JOHNSTON HAROLD MORE DATA ABOUT HIM
20010 DATA SCHMIDT  RALPH  MORE DATA ABOUT HIM
20020 DATA FLORBUSH TINA   MORE DATA ABOUT HER
20030 DATA ABRAMS   NIGEL  MORE DATA ABOUT HIM
20040 DATA RENKO    DIANE  MORE DATA ABOUT HER
20050 DATA ELLIS    KENT   MORE DATA ABOUT HIM
20060 DATA VINTON   JULIE  MORE DATA ABOUT HER
20070 DATA STANLEY  MYRON  MORE DATA ABOUT HIM
20080 DATA SWENSON  HELGA  MORE DATA ABOUT HER
20090 DATA JOHNSTON ALBERT MORE DATA ABOUT HIM
```

The preceding portion takes care of both the data definitions and the production of a dummy file which can be used to test the program. You will be producing your own data files for the future uses of the sort portions of this program.

```
300 FOR N = 1 TO 300:WHERE(N) = N:NEXT N
```

Set up the original values for the WHERE array. Now read the source file. Make it an endless loop for reading and putting away, stopping only if:

a. Exceed 300 data items, or
b. Hit end of file on source.

This time provide a separate error trap so that you can distinguish between an end-of-file and another error condition. After a

TRAP is "sprung," the error code will be put into location 195. You can get at it by doing a PEEK(195). If the value there is a 136, it means end of file. Other errors are explained in your *ATARI BASIC Reference Manual*. You may wish to treat other errors in different ways.

Here is the set of lines for reading the data file and the index file. Notice that the dummy file names are used. You will substitute your own file names later.

```
400 OPEN#2,4,0,"D:DUMMY.DAT"
410 OPEN#3,4,0,"D:DUMINDEX.DAT"
```

Open both data and index files for reading.

```
450 COUNT = 0:KEYPLACE = 1
```

Where in the input string should the data be placed?

```
460 TRAP 600
470 PRINT "READING: KEYS FROM SOURCE /
    INDEX "
```

Show the user what is happening now.

```
480 INPUT#2,DUM$
```

Get the whole record into memory. If the key is buried somewhere in the record rather than as the first X characters, then this will provide an easy way to get it out.

```
490 CUT$ = DUM$
```

Cut it down to 8 characters max Saves time.

```
500 KEY$(KEYPLACE) = "          "
```

Blank fill this section of KEY$ if it is possible that CUT$ is not already filled with blanks to the right of its actual data.

```
510 KEY$(KEYPLACE) = CUT$
```

Put away this piece of the string array.

```
530 COUNT = COUNT+1
540 KEYPLACE = KEYPLACE+8
```

Point to the next available entry in the KEY$ table.

```
550 IF KEYPLACE < 2400 THEN GOTO 480
```

Get the next record. The only possible exit is end-of-file or running out of room in the KEY$ data area (more than 300 records as currently set up). It can be increased if user has enough memory.

```
560 PRINT "TOO MANY RECORDS":END
570 PRINT "UNKNOWN ERROR":END
```

Now comes the error trap where we make sure that end-of-file has been reached:

```
600 IF PEEK(195) <> 136 THEN 570
610 PRINT "END OF SOURCE FILE"
611 PRINT:PRINT "STARTING SORT"
612 PRINT
613 CLOSE#2:CLOSE#3
```

The files get closed because you are finished getting the input for a while.

Next, set up the arrays which will be used to reference the key information. The program segment used here will be identical to that first presented as the sample of a string bubble sort, but it has been adapted to the key sort (also called a tag sort) function which was just discussed.

The line numbers and variable names have been kept the same, with the exception that most of the line numbers are exactly 1000 greater than in the original example.

```
1006 PINTER = 1
```

NOTE: The spelling here is PINTER instead of POINTER. This example emphasizes that you cannot use any form of ATARI BASIC keyword (in this example "POINT") unless it is used as a keyword, even if embedded within a variable name. The only time any keyword can be used otherwise is in a quoted statement (such as the subject of a PRINT or a string assignment), a REM statement, or a DATA statement.

```
1007 FOR MM = 1 TO COUNT
```

To save time, only initialize the number of items in these arrays
which are actually going to be used during the sort.

```
1008 START(MM) = PINTER:CHLAST(MM) =
     PINTER+8
```

The maximum length of the key in this example is 8 characters
as compared to 16 in the original example.

```
1009 PINTER = PINTER+8
1010 WHERE(MM) = MM
1011 NEXT MM
```

Now comes the part that is very similar to the string bubble sort
example:

```
1045 YES = 1:NO = 0
1050 FOR N = COUNT TO 2 STEP -1
1060 ANYSWAP = NO
1070 FOR P = 1 TO N-1
1075 S1 = START(WHERE(P+1))
1076 C1 = CHLAST(WHERE(P+1))
1077 S2 = START(WHERE(P))
1078 C2 = CHLAST(WHERE(P))
```

Use lines 1075–1078 to establish the character positions in the
string which are currently pointed to by the WHERE array.

```
1080 IF ARRAY$(S1,C1) >= ARRAY$(S2,C2)
     THEN 1120
```

If they are either greater than, or equal to, then no swap should
take place.

```
1081 REM STRING COMPARE REMAINS THE
     SAME.
1085 ANYSWAP = YES
1090 TEMP = WHERE(P+1):WHERE(P+1) =
     WHERE(P):WHERE(P) = TEMP
```

```
1091 REM SWAP THE TAGS (POINTERS TO THE
     KEYS)
1120 NEXT P
1130 IF ANYSWAP = NO THEN GOTO 2000
1140 NEXT N:REM DO NEXT PASS
```

Here is where the differences will begin. It is now necessary to somehow present the results of the sort to the user, and to save the results for future use.

The plan is to save the array called WHERE under the name WHERE.NDX on disk unit 1. If you wish, you might want to use some of the techniques shown earlier in this book to ask the user for the name of a file into which this index should be placed, then build a string name including "D1:", and so on, for putting the data away.

First it will be put away, then you will see how the data is used.

```
2000 OPEN#2,8,0,"D:WHERE.NDX"
```

Reuse IOCB number 2 because the file it used before was closed.

```
2010 PRINT#2,COUNT
```

Save data in the file to show how many records are expected. This way you won't have to worry about hitting end-of-file when it is read back in again.

```
2020 FOR N = 1 TO COUNT:PRINT#2,
     WHERE(N):NEXT N
```

Put the pointer array onto the disk.

```
2030 CLOSE#2
```

Close the file.

The following lines can be used to demonstrate to the user that the file has been sorted after all.

```
3000 OPEN#2,4,0,"D1:WHERE.NDX"
3010 OPEN#3,4,0,"D1:DUMINDEX.DAT"
3015 INPUT#2,NUMBEROFENTRIES
```

Keep the user informed.

```
3018 GR,0:POSITION 2,3:PRINT "READING
     RECORD # ";
```

Find out how many have to be read.

```
3020 FOR RECNO = 1 TO NUMBEROFENTRIES
```

Go after all of them.

```
3030 INPUT#3,S,B
3040 STARTS(RECNO) = S : STARTB(RECNO)
     = B
```

For this kind of work, where the disk or some other device will be very busy, you will want to assure the user that things are really happening. Therefore, add lines such as 3018 and 3050 to keep the user informed.

```
3050 POSITION 20,30:PRINT RECNO
```

Get the WHERE array back in here as though reading it for the first time.

```
3060 INPUT#2,WH:WHERE(RECNO) = WH
3070 NEXT RECNO
```

You may want to put in some error traps for end-of-file, but in this case, the number of records is known, so it might not be needed.

```
3080 CLOSE#2:CLOSE#3
```

You will not need source data any more, now that the index data is in memory.

Now you have both the revised pointer list (WHERE) and the pointers to the start of each record (STARTB, STARTS). Refer to the program in Chapter 4 called FASTSRCH.DMO. You will see in the next program segment that the POINT command is used in the same way to go directly to the record number which is requested.

The records are printed to the screen in alphabetical order . . . with one exception (JOHNSTON), which will be discussed.

```
3500 OPEN#2,4,0,"DUMMY.DAT"
```

Open the original data file to read the complete record.

```
3510 FOR N = 1 TO NUMBEROFENTRIES
3520 POINT STARTS(WHERE(N)),STARTB
     (WHERE(N))
```

Point the source file to the correct starting position.

```
3530 INPUT#1,DUM$
```

Get the record you want (random access into a sequential file, access by alphabetical order).

```
3535 PRINT DUM$
```

Print it.

```
3540 NEXT N
```

Get them all.

```
3600 END
```

You will notice above that there is still one item out of alphabetical order. That is the two JOHNSTON records. This is because you have only provided, in this sample program, the sort of the first 8 characters of the whole file.

Occasionally, you might want to do what is called a *multiple key sort*. This means that you fully expect that within the first set of keys, one or more will be identical (JOHNSTON in our case), and you want them *all* in the absolute correct order.

To do this, once the first key sort has been completed, you already have a large head start on the secondary (and third level, fourth level, etc.) sort. Here is why.

During the first level sort, you always swapped items only if the second one was *less than* the first one. Now that the first level is done, you would only need to refer to the completed WHERE array and to look at any items in the first key which are identical. It is only for those that the second key must be used.

In the example used, WHERE(4) points to record 1, one of the

JOHNSTON's, and WHERE(5) points to the other one at record number 10.

So if you found these two source strings to be identical, you might consider reading a second key (user directed, of course), which would help to break the tie. The technique itself won't be discussed here, since the other parts of the program can be adapted to include this process if you wish.

You will probably want to do the printing of the records as part of a data entry/editor program, where once the record has been presented, the user can modify selected fields, then put the modified record back where it came from. For this kind of activity, you can still use the same record-pointing approach, but you should probably use two IOCBs for the source file. One of them would be opened for reading the source file and the other would be opened for writing a revised version of the file (as demonstrated for RAW.DTA and EDITED.DTA).

Or, as an alternative, a single IOCB may be opened for both reading and writing. This, however, is more tricky, since you must be sure the *pointers* to the file position are in exactly the right place, or you can destroy your incoming data file.

REVIEW OF CHAPTERS 5 AND 6

1. A bubble sort is a sort in which items adjacent to each other are compared (next sequential item in the list). If the second is less than or equal to the first item in any given sequence, the items are swapped. This continues pass after pass with one less item at the end of the list per pass until all items are in order.
2. An insert sort takes each item in a list, one at a time, and compares each to all previously examined items in the list, to see where it should be *inserted* into the list. In this form of sort, there is one more, rather than one less, items to compare on each pass.
3. Either the bubble sort or the insert sort can be shortened by examining certain swap criteria before starting over at the beginning of the list for another pass.

4. These sorts can be applied to *keys* of a long set of records, where the pointers to the records, called *tags,* are swapped rather than the keys themselves. This provides revised tag file which can be used to quickly reference the original data file.

5. A combination of a data file creator program and the sort program can give you the ability to create, then quickly reference, any data record, without rewriting the original records in the new sequence.

CHAPTER 7

Getting Directly Into the Screen Data

Up to this point, all of the programs have somehow used the Screen Editor to put data onto the screen. This chapter will concentrate, instead, on going directly into the ATARI memory system either to place or retrieve data. The primary goal of this chapter is to show you how each of the graphics screens is produced, and provide you with enough information to build your own custom displays.

To this end, this chapter will show you, step by step, how to build a primitive but useful graphics-screen creator. You will be able to put away the screens which you create and call them up as part of the other programs which you write.

HOW THE ATARI SCREEN IS FORMED

The television screen, although it may look like a continuous picture, is actually composed of a large number of dots horizontally and vertically. Normally they are so close together that they do appear to form a continuous picture.

The horizontal dots are actually each one subdivision of a television scan line. It is the brightness of the scan line (and/or the color) which is changed as the television beam sweeps across

the screen. This is what separates each horizontal line into the series of dots.

There are a total of 525 horizontal lines on the U.S. television screen, counting from top to bottom for each single frame in a TV picture. However, each frame of 525 lines is actually made from two *fields,* and each field uses 262.5 lines. This is called *video interlace* because one field supplies the odd lines while the other has the even numbered lines. This is very important to normal TV pictures. However, the static display of a computer essentially repeats the same 262.5 lines on both fields that *interlace* to make the single *frame* of a video picture. Because of the repetition of field information, the ATARI system typically uses the center group of 192 (really 384) of the horizontal lines out of the total of 262.5 (really 525). This guarantees that the display which the computer produces will be somewhere in the visible central part of the screen. This accounts for the vertical part of the display.

For the horizontal part of the display, the maximum number of light-dots which the ATARI computer can show is 384 (this is called *wide playfield mode*). But of this total number of possible positions, only about 320 will be within a viewable central screen area. (This is called a *normal-width playfield.*)

Therefore you will normally see, in your *ATARI BASIC Reference Manual,* that the maximum graphics *resolution,* the number of individually addressable dots across and down, is 320 by 192.

In order to form a display, the special hardware within the ATARI computers subdivides the screen into a number of horizontal stripes, stacked on top of each other. There may be as few as 12 of these stripes (for example, when GRAPHICS mode 2 + 16 is used . . . 12 lines of 20 characters each), or there may be as many as 192 stripes (when GRAPHICS mode 8 + 16 is used . . . 192 lines from top to bottom, each with 320 addressable graphics points in a single color).

ATARI BASIC provides the user with the capability of forming various kinds of display screens, simply by using one of the GRAPHICS commands. For example, GRAPHICS 0, or GR.0, forms a clear new screen composed of 24 lines of text material. GRAPHICS 2 forms a clear new screen using 10 lines of 20 characters each, with a text area of 4 lines below that section, and so forth.

What you will find in common among all of these various BASIC callable modes is that a total of 192 lines will be used for the display, from top to bottom.

This number, 192, is very important. If more than 192 lines are defined, there may be some difficulty in allowing the central processor enough time to finish some of its jobs. These jobs are only done once it appears that the display time is finished and before a new display is to begin. This could cause *loss of sync,* which in turn will cause the picture to roll or jump.

Let's look at how these available graphics modes add up to 192:

- GR.0 is 24 lines of text. Each line of text is 8 horizontal lines high (24 × 8 = 192).
- GR.1 is 24 lines of wider text. Each line is 8 horizontal lines high, as the above.
- GR.2 is 10 lines of wider and higher text, plus 4 lines of normal text.

  Each of the 10 big text lines is 16 horizontal lines high (160 lines total), plus the 4 text × 8 lines (another 32) for a total of 192.

  all the way down to
- GR.8 + 16 (which you can also call GR.24 if you wish), which has 192 lines.

Hopefully, by these examples, you can see that the screen is really composed of a sequence of horizontal stripes of various widths, stacked on top of each other to form the complete display.

This part of the screen will be called the *playfield.* This playfield part is relatively stationary.

There are also other objects which can be moved across the screen, but this chapter covers only the playfield portion of the display.

HOW THE DISPLAY IS MANAGED

There is a separate display processor within the ATARI computers. Its job is to produce the playfield and the movable objects on the screen.

Since there are many different kinds of displays which are possible on this system, this processor must be told exactly how the screen is to be formed. It will then do the job it has been given in a continuous loop until other instructions are received.

The display processor is called *ANTIC*. It operates at the same time as the central processor, and slows the processor down, when necessary, to do its own job. You see, the job of the display processor is more important than the job of the central processor. This is because you want to see a stable picture, with no *jumping* or *tearing*. Therefore, the display processor must always be allowed to do its job so that the display will remain stable.

The display processor receives its instructions in the same way the central processor receives them; that is, they are stored in memory.

The central processor has many different kinds of instructions, but the display processor has only a few. The major kinds of instructions which it executes are:

1. Define a set of lines in a specific graphics mode.
2. Jump to another part of the memory for new instructions.
3. Jump to the start of the instruction group to wait for the start of a new screen display time.

There are other bits within the instruction which affect what will happen on the screen; however, these other things do not affect the fact that the primary instruction is still one of the three just mentioned.

In the section which follows, you will see a small program in ATARI BASIC that is supposed to represent what the display processor is doing. The techniques used in this BASIC program will be used as the basis for the program which creates and edits the screen.

Here is a representation of how the display processor operates:

```
NEW  RETURN
10 DIM A$(38),B$(1000)
11 REM SAVE SOME SPACE FOR THE DATA
15 A$ = "THIS DEFINES THE DATA FOR
   LINE........"
```

```
16 REM 38 TOTAL CHARACTERS BETWEEN THE
   QUOTES
100 FOR N = 1 TO 19
110 PRINT CHR$(125):REM CLEAR SCREEN
120 LNG = 1
130 B$ = "":REM SET INITIAL LENGTH TO
    ZERO FOR DEMO
140 FOR M = 1 TO N
150 B$(LNG) = A$
155 REM BUILD A SCREEN OUT OF
    PREDEFINED DATA
160 B$(LNG+32,LNG+33) = STR$(M):REM
    CONVERT LINE NUMBER INTO A STRING
    VALUE AND STORE IT.
170 B$(LNG+37 = ".":REM RESTORE PROPER
    LENGTH
175 LNG = LNG+38:REM POINT TO DATA AREA
    FOR NEXT LINE.
180 NEXT M
190 PRINT B$:REM SHOW THE SCREEN THAT
    WAS BUILT
200 FOR Q = 1 TO 300:NEXT Q:REM DELAY A
    LITTLE
210 NEXT N
```

If you run this program, you will see the screen being made longer, one line at a time. This is the same technique, slightly modified, that will be used in the screen-builder program.

When you wish to tell the ANTIC to assign one segment of the screen to be one of the 14 possible graphics modes, it will need two pieces of information:

1. Which one of the modes to use.
2. Where is the data which is to be shown on the screen for this mode.

There are two kinds of instructions which define the kind of display lines to use:

1. Use this mode, and find the data in the memory after the data defined for the previously defined line.

2. Use this mode, and look for the display data at the location which is specified with this instruction itself.

In the first simulation example, the first method was used. In other words, the data for the second line was found in the B$ immediately following the data for the first line of the display. B$ is an array of many (here 1000) memory locations, all in a row. This, then, makes the situation similar to the first method.

The only thing that you must remember when using this form of the instruction is that somewhere preceding this instruction, there must have been an instruction "type 2" which actually initializes the memory pointer. Actually, in the preceding example, there is one instruction of "type 2" already present. That is, the print statement for the first line which said PRINT B$. When you told it about B$, it knew where the data for the first line would be taken from, and in the continued printing, knew where all of the data for subsequent lines would be located (after line 1).

Just to clarify this principle a bit further, let's look at a modified example of the earlier program in which all of the display instructions are of "type 2." That is, the system will be told explicitly, for each kind of display mode grouping (here for each display line in graphics mode 0), *and* exactly where to find the data for that line.

```
10 DIM TEMP$(32):REM DEFINE AN INPUT
   STRING
15 DIM ANS$(1)
20 DIM A$(32),B$(32),C$(32),D$(32)
30 TEMP$ = ",":TEMP$(38) =
   ",":TEMP$(2,LEN(TEMP$)) = TEMP$
31 REM A NEAT WAY TO FILL AN ENTIRE
   STRING WITH THE SAME CHARACTER
40 A$ = TEMP$:B$ = TEMP$:C$ = TEMP$:
   D$ = TEMP$
41 REM INITIALIZE THE "SCREEN DISPLAY
   MEMORY"
50 PRINT CHR$(125):REM CLEAR THE SCREEN
60 PRINT "A →";A$
70 PRINT "B →";B$
```

```
80  PRINT  "C  →";C$
90  PRINT  "D  →";D$
100 PRINT:PRINT "USER INPUTS A LINE OF
    DATA BELOW"
110 PRINT:PRINT " →";INPUT TEMP$
120 POSITION 2,9
130 PRINT "PUT INTO SCREEN LINE
    (A/B/C/D)";
140 INPUT ANS$
145 IF LEN(ANS$) = 0 THEN 120
150 WHICH = ASC(ANS$)-ASC("A")+1
151 REM A-A = 0,B-A = 1,C-A = 2,
    D-A = 3, SO IF ADD 1, OK VALUE
155 IF WHICH<1 THEN 120
160 ON WHICH GOTO 210,220,230,240
170 GOTO 120:REM IF NO ACCEPTABLE
    ANSWER
210 A$ = TEMP$:GOTO 50
220 B$ = TEMP$:GOTO 50
230 C$ = TEMP$:GOTO 50
240 D$ = TEMP$:GOTO 50
```

This program demonstrates a manner in which each of the lines of data defined for the display processor can have its own data area reserved. If you change what is located in that data area, you change what appears on the display itself.

This is the alternate method to that of defining a large memory area, and simply telling the processor to continue using the memory where the previous line left off. In the sections which follow, you will begin to get away from the display simulation and into the actual building of a custom display on the screen.

BUILDING A CUSTOM DISPLAY

The next program segment will simply build one custom display. It will consist of 4 lines of 40-character text, and 10 lines of 12-character (very large) text. The custom version which is to be built will be the *reversed* version of normal GRAPHICS MODE 2.

In fact, if you are only marginally familiar with this particular mode, try the following program first. It will demonstrate MODE 2. Then the custom build program will take the text area and put it on the top with the graphics area on the bottom.

```
10  REM GRAPHICS 2 DEMO
20  GR,2
30  PRINT#6;"THIS IS GR,2"
40  OPEN#1,4,0,"K:":REM PATH FOR
    NO-RETURN KEY
50  PRINT "PRESS ANY KEY FOR NORMAL
    DISPLAY";
60  GET#1,M:REM WAIT FOR ANY USER
    KEYPRESS
70  GR,0:END:REM RESTORE SCREEN TO
    NORMAL
```

If you run that program, you will see how the normal GRAPHICS 2 display appears. Now the program below will reverse the relative positions of the two kinds of displays. You will see that some POKE statements have been used instead of PRINT statements. This is to introduce you to the direct access to the screen data, rather than using the Screen Editor to do it.

The custom display will be built by the following instruction sequence:

1. Use small text (40 char/line). (Data for this first line is located at a specific memory location.)
2. Use small text for next 3 lines. (Data for these lines follows the data for the first of these lines.)
3. Use largest text for the next line. (Data for this large text should start at some specific memory location.)
4. Use largest text for the next 9 lines. (Data for these lines follows the data for the first of these lines.)
5. End the screen display here by jumping to the beginning of this list and waiting for the next time a screen is to begin so we can do it again.

What you have just seen is called a *display list*. It is the sequence of instructions which the display processor is to perform.

It is this sequence that you will now see translated into machine code instructions for the display processor.

```
10  XMEM=PEEK(106)
```

This will set the value of X to the current top of memory. Since the display shares the memory with the central processor, it will be necessary to set aside some space for exclusive use by the display processor.

```
20  LOWX=XMEM-16:POKE  106,LOWX
25  LOWXPNT = LOWX * 256
```

This saves about 4096 memory locations for use by the display.

```
30  GR.0
```

The next line is used to move the current *normal* display list into the area of free memory. You see, when BASIC wants to form a display area, some of the memory at the top section gets reserved for the display processor to use. Then BASIC knows that it will be free to use any of the rest of the memory from its low limits (just above the DOS) to its highest limits (just below the display area). It uses the contents of location 106 to tell it what is the highest possible memory area into which it can write.

Location 106 is written by the power-on routine, which figures out how much memory is active long before BASIC starts up in the system. ATARI BASIC normally only reads location 106, and this tells how much memory it can work with. If you change the value, as has been done above, you have reserved this memory for your own personal use and BASIC won't bother it even after a program ends.

NOTE: **SYSTEM RESET** restores this to its original value.

Now what will be done is to reserve some space for the display list in the same way that ATARI BASIC does it. In particular, ATARI BASIC combines the area for the display list and the data into one single space. The following diagram is called a *memory map,* and it shows in a very simplified form how the memory space is divided:

```
+ ------------------ +          Top of memory area
                                (address = 65535)
        Operating
        System

+ ------------------ +          RAMTOP where location 106
                                normally points
      Display Data

+ ------------------ +
      Display List
+ ------------------ +          Top of BASIC random access
                                memory (RAM) area (includes
      BASIC RAM                 user program)

+ ------------------ +
        DOS
    and Operating
    System RAM
+ ------------------ +          Bottom of memory
                                (address = 0)
```
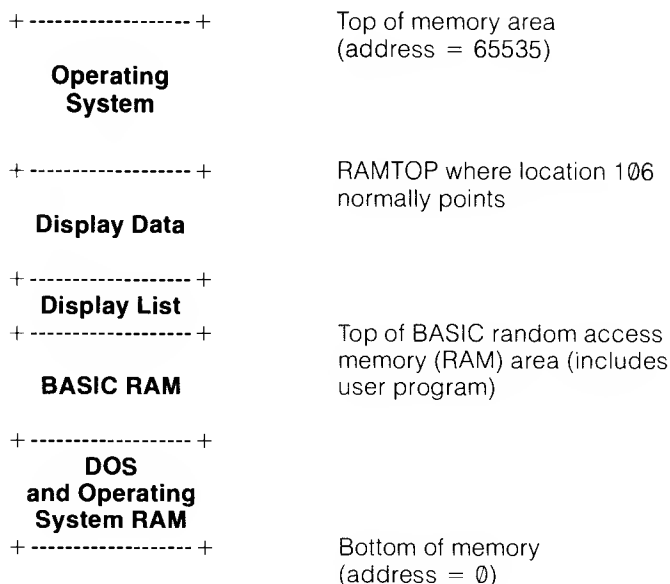
Let's reserve 300 spaces for the display list. This will be more than enough for most applications.

```
40  DS = LOWXPNT+300:REM DS MEANS DATA
    AREA START
```

Now begin assigning the different pieces of the display list. The values will be generated by a subroutine so that the routines can be easily transported into other programs.

To do this, it will be necessary to establish a certain set of values or tables which will be useful throughout these routines. The first pointer will be the pointer to the area where the display list is being built. Call it DLPNT. It starts at the beginning of the area called LOWX.

```
50  DLPNT = LOWX:REM NEW DISPLAY LIST
    POINTER
```

A set of values will also be mentioned here for the first time, but not discussed until later. These are the values for horizontal scrolling and vertical scrolling.

```
60 HSCROLL = 0:VSCROLL = 0
```

Now, before any form of display list routine may be formed, you will need a set of tables which will contain definitions for each of the different kinds of graphics modes. Here it is:

```
6 DIM MEMPERLINE(15),LINESPERMODE(15)
```

These lines save space for an array showing how much memory is needed (minimum) for each one of the ANTIC modes which you can define, and how many scan lines each of the modes will take up.

The first of the arrays contains numbers indicating, for example, how many characters there can be in a single line, or how many memory locations it takes per line to represent a certain kind of graphics mode.

The second array is related to that magic 192 number mentioned earlier. You will notice next that, for example, ANTIC MODE 2, which corresponds to ATARI BASIC GRAPHICS MODE 0, takes up 8 lines each time it is placed on the screen. Therefore, 24 lines of text times 8 video lines per text line gives the 192 figure mentioned earlier. As a display list is built, this value will be used to keep track of how far toward the 192 you may be.

```
30000 DATA 40,40,40,40,20,20,40,80,80
30010 DATA 160,160,160,160,320
```

This is the data for the minimum number of characters per line (in a "normal playfield width" mode).

```
30020 DATA 8,10,8,16,8,16,8,4,4,
      2,1,2,1,1
```

This is the data for the number of lines that each of the modes uses. In the initialization loop shown next, you will see that these data items represent the data for ANTIC MODES 2 through 15. There is *no* ANTIC MODE 1.

```
80 FOR N = 2 TO 15
82 READ M:MEMPERLINE(N) = M:NEXT N
84 FOR N = 2 TO 15
86 READ M:LINESPERMODE(N) = M:NEXT N
```

A final set of initialization items:

```
90 TRUE = 1:FALSE = 0
91 BADMODE = 1:TOOMANY = 2:OUTOFMEM = 3
92 REM DEFINE POSSIBLE FAIL CODES
95 NORMALWIDE = TRUE
96 REM DOING A NORMAL SIZE SCREEN AT
   FIRST
```

And now, the subroutine that will place an item into the display list and update the pointers to the area where both the list and the data are being stored will be reviewed. This routine assumes that there is already a dummy display list formed in the area which is being written so that the display being formed this way can be immediately shown.

```
25000 REM ENTRY VARIABLES: MODE = THE
      MODE TO USE
25010 REM NORMALWIDE = TRUE IF VALUES
      FROM MEMPERLINE ARRAY ARE TO BE
      USED
25015 REM MEMPERLINE(1) = HOW MUCH
      MEMORY TO USE IF NORMALWIDE =
      FALSE
25020 REM HSCROLL AND VSCROLL ARE FALSE
      IF NO HORIZONTAL OR VERTICAL
      SCROLLING IS NEEDED
25030 REM ROUTINE STARTS BELOW
26000 IF MODE < 2 OR MODE > 15 THEN
      OK = FALSE:REASON = BADMODE:
      RETURN
26010 DINSTRUCT = (MODE+64) +
      (HSCROLL*16) + (VSCROLL*32)
26020 CTEMP = COUNT+LINESPERMODE(MODE)
26030 IF CTEMP > 192 THEN OK =
      FALSE:REASON = TOOMANY:RETURN
26040 IF NORMALWIDE = FALSE THEN 26060
26050 DSTEMP = DS+MEMPERLINE(MODE):GOTO
      26070
```

```
26060 DSTEMP = DS+MEMPERLINE(1)
26070 IF DSTEMP > (XMEM*256)THEN OK =
      FALSE:REASON = OUTOFMEM: RETURN
26080 POKE DLPNT,DINSTRUCT
26081 REM PUT AWAY THE INSTRUCTION
26090 DHIGH = INT(DS/256):DLO =
      DS-DHIGH*256
26091 REM SEPARATE VALUE OF DATA
      POINTER INTO A LOW PART AND A
      HIGH PART
27000 POKE DLPNT+1,DLO:POKE
      DLPNT+2,DHIGH
27001 REM STORE THE ADDRESS OF THE
      START OF DATA AREA FOR THIS
      DISPLAY LINE
27010 DLPNT = DLPNT+3:REM NEXT
      AVAILABLE AREA
27015 GOSUB 28000:REM INITIALIZE THIS
      AREA
27020 COUNT = CTEMP:DS = DSTEMP:OK =
      TRUE:RETURN
```

Line 26000 looks to see if the display mode you want to use is between 2 and 15, since these are the only modes that the ANTIC can use.

Line 26010 calculates the instruction word. It is composed of the horizontal scroll value, the vertical scroll value (these will be used later in the chapter), and the mode. The reason for the value of 64 here is that there are two ways to represent an ANTIC instruction.

One is to say "use this mode for the next set of display lines." This kind of instruction does not require the *address* information showing where to find the data for this set of lines. In other words, the ANTIC is told by this kind of instruction (mode number only) that it should simply use the continuation of the previously defined data area to find the data for the mode it is now trying to display.

The other way, which is being used here, is to ask the ANTIC

to "use this mode *and* find the data in a specific area of memory."
The 64 adds this extra requirement to the instruction. It is called
the *Load Memory Scan* bit. This means that the next two memory
locations in the display list must contain the pointer to where the
data, for this display piece, is stored. The pointer is stored with
its low part first, then its high part as shown previously.

Line 26010 does not include one extra mode which is possible.
This is the *Display-List Interrupt* bit. If it is on, the display proces-
sor (ANTIC) will *interrupt* the main processor and force it to do a
defined extra task before it can go back to doing what it was
doing before. This interrupt happens each time the ANTIC en-
counters this instruction. It is this feature that enables the "rolling
rainbow" effect you sometimes see in ATARI games.

This concept is more advanced than was intended for this
book and will not be covered here. However, if you wish to pursue
it further, you can generalize line 26020 to include it by adding
the term:

```
, , , , ,  +  ( DLIBIT * 128 )
```

where the variable DLIBIT will either be a 1 or a 0 (TRUE OR
FALSE).

Lines 26030 and 26040 check to see that the video lines which
are currently being added to the display list don't make it longer
than 192 lines. If it is acceptable, it updates the line count.

Lines 26040–26070, and thereafter, adjust the data area point-
ers. This basically says that this memory area has been *allocated*
and the pointer must now point to the next available memory
space. In this area, it also checks to see if there was enough
memory to define this mode in the first place.

Line 26080 puts away the instruction value and updates the
value of the display list pointer since we just added three items
(instruction, low address, high address) to the display list.

Line 27015 is placed in this routine so that there will be some
kind of value put into the memory space for each of the graphics
modes. You see, if you have just applied the power, the ATARI
computer will either place 0s into this upper memory area which
you are now using, or perhaps some other pattern, depending
on whatever previous use was made of this area for display, and

so forth. Line 27015 calls another subroutine which puts what is called a *default value* into each of the memory locations just assigned to this latest display mode so that when you view the new screen, there will be something predictable for you to look at.

Finally, line 27020 adjusts all of the pointers because everything else looked OK and allowed the writing of a new item into the display list. This completes the display list updating subroutine.

Now for the initialization routine, very short and easy:

```
7 DIM DFAULT(15)
88 FOR N = 2 TO 15: READ M: DFAULT(N) =
   M:NEXT N
30030 DATA 45,45,69,69,85,85,85,85,
      85,85,85,85,85,85
28000 M = DFAULT(MODE)
28010 FOR N = DS TO DSTEMP-1
28020 POKE N,M
28030 NEXT M
28040 RETURN
```

which fills the new data area with the default data.

In the 28000 subroutine, it said that this routine assumed that there was a dummy display list built up, and that it was this dummy whose pieces were being replaced. Let's look at what the dummy display list must contain to fulfill this requirement.

First, think back to the first example given in this chapter. You may recall that the display there was built one line at a time, taking up space on the screen starting at the top, then proceeding down the screen to the point where it stopped.

You might imagine this sequence of instructions, instead of being constructed as they were originally, being constructed as:

```
TOP      PRINT A LINE
         PRINT A LINE
         PRINT A LINE
         GOTO TOP
         GOTO TOP
```

```
GOTO  TOP
GOTO  TOP
GOTO  TOP
```

If it is executing instructions from the top down, then once it has printed all of the lines, it encounters the first GOTO TOP instruction. All of the other GOTO instructions would never get executed, since the first one it finds sends it up to the top of the program again. In this example, then, the rest of the GOTO TOP instructions are dummies. That is, they are simply there to take up space.

It is quite convenient, in the case of the display processor instructions, that the "jump and wait for vertical blank" (the next start of the display time) instruction takes up exactly the same amount of space as the "use this mode from this new address area" instruction (LMS bit is on). This means that for each of the cases where we simply want to extend the length of the current display list, it is only necessary to replace one of the JUMP instruction groups. An example is shown next.

```
TOP   DISPLAY 8 BLANK LINES
      DISPLAY 8 BLANK LINES
      DISPLAY 8 BLANK LINES
      JUMP TO TOP  ← REPLACE WITH: DISPLAY MODE 2
                                   USING  DATA  AT
                                   "DS"

      JUMP TO TOP
      JUMP TO TOP
      JUMP TO TOP    (Last 2 and any others beyond here
                      are dummies and can never get ex-
                      ecuted until prior jumps are replaced
                      by display-type of instructions.)
```

As you can see, if the instructions for either JUMP or DISPLAY take up the same amount of space, then the *display list can always be made to end with a JUMP instruction.*

Now, here is the section of the program which will set up this dummy display list, which you will modify.

Actually this program will not do exactly what you have seen

earlier. Instead, it will perform one extra instruction before it jumps to the top of the display list.

This extra instruction will provide a status line on the screen. As each of the display list lines is added, the status line will be moved one step toward the bottom of the screen.

Here is the program. The explanation for this program follows the listing:

```
405 DIM A$(38)
420 GPOS = LOWXPNT+2:REM START POINT
    FOR DLIST
430 FOR N = 1 TO 3:POKE GPOS,112:GPOS =
    GPOS+1:NEXT N
431 REM EACH 112 IS A "DISPLAY 8 BLANK
    LINES"
440 JMPAD = GPOS+585:REM PUT JUMP
    ADDRESS NEAR END OF ALLOCATED
    DISPLAY LIST AREA (IS 600 MAX LONG)
445 JMPHI = INT(JMPAD/256):REM SEPARATE
    INTO HI AND LO ADDRESSES
450 JMPLO = JMPAD - JMPHI*256
470 FOR N = 1 TO 192:REM NOW DEFINE 192
    SETS OF JUMP TO SHOW STATUS LINE,
480 POKE GPOS,1 :REM THIS SAYS JUMP
490 POKE GPOS+1,JMPLO
500 POKE GPOS+2,JMPHI
510 GPOS = GPOS+3:NEXT N
511 REM EACH OF THE 192 BECOMES A "JUMP
    TO STATUS"
530 GPOS = JMPAD: NOW FORM THE STATUS
    LINE DISPLAY
540 POKE GPOS,66:REM MODE 2 (40 CHAR
    LINE)
550 DLST = LOWXPNT+600:REM DATA AREA
    BEGINS 600 LOCATIONS PAST START OF
    RESERVED AREA
560 HIAD = INT(DLST/256):LOWAD =
    DLST-256*HIAD
```

```
561  REM CALCULATE LOW AND HI PART OF
     JUMP ADDRESS
570  POKE GPOS+1,LOWAD:POKE GPOS+2,HIAD
580  GPOS = GPOS+3
590  POKE GPOS,65:REM JUMP AND WAIT
     VERTICAL BLANK
591  POKE GPOS+1,2:REM WHERE DISPLAY
     LIST STARTS
595  POKE GPOS+2,LOWX:REM HI PART OF
     WHERE DLIST STARTS
600  POKE 560,2:POKE 561,LOWX
601  REM 600 TELLS ANTIC WHERE TO FIND
     THE NEW LIST
```

This program segment builds a display list which consists of:

```
DISPLAY 8 BLANK LINES
DISPLAY 8 BLANK LINES
DISPLAY 8 BLANK LINES
   JUMP TO STATUS LINE
   JUMP TO STATUS LINE
 ...lots more of those...
(describing max of 192 lines)


DISPLAY ONE LINE OF 40-CHARACTER TEXT
WHOSE DATA STARTS AT A FIXED LOCATION
JUMP AND WAIT FOR VERTICAL BLANK
```

This type of program, then, will produce a display list which has, as it begins, an entirely blank screen with the status line as the only line showing at the top of the display.

As each new line definition is added, the new definition replaces one jump to the status line display instruction. This forces the status line to move down on the screen, one step at a time. You can also move the status line up one segment on the screen simply by replacing the last-added text or graphics line definition with a jump to the status line.

As an example, you might be building a display list consisting of:

```
DISPLAY 8 BLANK LINES
DISPLAY 8 BLANK LINES
DISPLAY 8 BLANK LINES
DISPLAY ONE LINE OF 20 COLUMN (LARGE)
  TEXT
DISPLAY ONE LINE OF 20 COLUMN TEXT
DISPLAY ONE LINE OF HIGH-DENSITY
  GRAPHICS (GR.8)
  JUMP TO STATUS LINE
  JUMP TO STATUS LINE
...MORE...
```

You may decide that it is not yet time to start the display of the high-density graphics. To restore the list to its original condition, you will only need to replace that high-density graphics description with a jump to the status line. The status line will then move up one on the screen, thereby erasing the graphics line you did not want.

Let's look at what the status line will contain, and how you will go about building the screen. Here is the status line:

```
Add Delete Edit eXit MODE=__
```

where the A, D, E, and X will be presented in reverse video (dark letters against a white background).

Here is what the different modes will mean:

Add— Add a new graphics mode area in the position now occupied by the status line. Push the status line down one position. Put some data there so that the user can see that this new line has been added. Data can be changed during edit mode.

Delete—Delete the graphics line just added and currently located above the status line. Move the status line up to the next position.

Edit— This mode now allows you to modify the sections of the screen which you have already defined. Normally you will define the whole screen, then go into edit to do something with the data.

eXit— Give the user the chance to save this display list somewhere.

understandable, nothing has been done to condense it or to make it fast. If you find it useful, then you may wish to find ways to speed up its operation. Following the program explanation, you will find other suggested improvements which you may decide to add to the program. Here is the combined listing:

```
6 DIM MEMPERLINE(15),LINESPERMODE(15)
7 DIM DFAULT(15),CUR(192)
10 XMEM=PEEK(106)
20 LOWX=XMEM-16:POKE 106,LOWX
25 LOWXPNT=LOWX*256
30 GRAPHICS 0:PRINT "WORKING..."
40 DS=45+LOWXPNT+600
50 DLPNT=LOWXPNT+5:REM NEW DISPLAY LIST
   PNTR
55 LINPNT=1:LINCNT=0:EDPNT=DLPNT
60 HSCROLL=0:VSCROLL=0
70 GOSUB 23000
80 FOR N=2 TO 15
82 READ M:MEMPERLINE(N)=M:NEXT N
84 FOR N=2 TO 15
86 READ M:LINESPERMODE(N)=M:NEXT N
88 FOR N=2 TO 15:READ M:DFAULT(N)=M:
   NEXT N
90 TRUE=1:FALSE=0
91 BADMODE=1:TOOMANY=2:OUTOFMEM=3
92 REM DEFINE FAIL CODES
95 NORMALWIDE=TRUE
96 REM DOING A NORMAL SCREEN AT FIRST
97 COUNT=0:MEMPERLINE(1)=40:REM DEFALT
300 OPEN #1,4,0,"K:"
310 TRAP 660
405 DIM A$(38)
420 GPOS=LOWXPNT+2
430 FOR N=1 TO 3:POKE GPOS,112:
    GPOS=GPOS+1:NEXT N
440 JMPAD=GPOS+585
445 JMPHI=INT(JMPAD/256)
```

```
450 JMPLO=JMPAD-JMPHI*256
470 FOR N=1 TO 192
480 POKE GPOS,1
490 POKE GPOS+1,JMPLO
500 POKE GPOS+2,JMPHI
510 GPOS=GPOS+3:NEXT N
530 GPOS=JMPAD
540 POKE GPOS,66
550 DLST=LOWXPNT+600
560 HIAD=INT(DLST/256):LOWAD=
    DLST-256*HIAD
570 POKE GPOS+1,LOWAD:POKE GPOS+2,HIAD
580 GPOS=GPOS+3
590 POKE GPOS,65:POKE GPOS+1,2
595 POKE GPOS+2,LOWX
600 POKE 560,2:POKE 561,LOWX
660 A$=" Add Delete Edit
    eXit               "
661 A$(36)=STR$(192-COUNT)
662 IF COUNT>92 THEN A$(38)=" "
663 IF COUNT>182 THEN A$(37)=" "
670 GOSUB 24000
710 GET #1,M
715 IF M=65 THEN 750:REM ADD
720 IF M=68 THEN 800:REM DELETE
725 IF M=69 THEN 900:REM EDIT
730 IF M=88 THEN 3000:REM EXIT
735 GOTO 710
750 A$="    Add Delete Edit eXit
    MODE=___           "
751 A$(36)=STR$(192-COUNT)
755 GOSUB 24000
756 GET #1,M:IF M<49 OR M>57 THEN 756
757 IF M>49 THEN MODE=M-48:GOTO 762
758 POKE DLST+30,17:REM PUT A 1 THERE
759 GET #1,M
760 MODE=M-48+10:POKE DLST+31,MODE+6
762 TRAP 660:REM TRAP ERROR TO REDRAW
```

```
765 IF MODE<2 OR MODE>15 THEN 660
770 GOSUB 26000
771 LINCNT=LINCNT+1:REM ADDED A LINE
775 IF OK=TRUE THEN 660
780 GRAPHICS 0:PRINT "FAILCODE =
    ";REASON
800 IF LINCNT=0 THEN 660:REM DONT BACK
    UP TOO FAR
801 DLPNT=DLPNT-3
802 CLAIM=PEEK(DLPNT)
803 COUNT=COUNT-LINESPERMODE(CLAIM-64)
804 A$(36)="    "
805 POKE DLPNT,1
810 POKE DLPNT+1,JMPLO
815 POKE DLPNT+2,JMPHI
818 LINCNT=LINCNT-1
820 DS=DS-MEMPERMODE(CLAIM-64):REM
    RECLAIM DATA MEMORY ALSO
830 GOTO 660
900 IF LINCNT=0 THEN 660:REM CANT EDIT
    IF NO LINES THERE!
905 A$="  M=  L=   C=   V=     NewV,
    Quit"
910 GOSUB 24000
915 DLP=EDPNT+(LINPNT-1)*3
920 MODE=PEEK(DLP)-64
925 C=CUR(LINPNT)+1
930 L=LINPNT
935 DV=256*PEEK(DLP+2)+PEEK(DLP+1)+C-1
940 V=PEEK(DV)
945 A$="  M=  L=   C=   V= ←  →   ↑   ↓
    NewV, Quit"
950 A$(5)=STR$(MODE):A$(9)=STR$(LINPNT)
955 A$(14)=STR$(C):A$(19)=STR$(V)
957 A$(38)="  "
960 GOSUB 24000
970 GET #1,M
975 IF M=43 THEN 1000
```

```
980 IF M=42 THEN 1100
985 IF M=45 THEN 1200
990 IF M=61 THEN 1300
995 IF M=78 THEN 1400
998 IF M=81 THEN 660
999 GOTO 970
1000 IF CUR(LINPNT)=0 THEN 940
1005 CUR(LINPNT)=CUR(LINPNT)-1:GOTO 915
1100 IF CUR(LINPNT)=(MEMPERLINE
     (MODE)-1) THEN 970
1105 CUR(LINPNT)=CUR(LINPNT)+1:GOTO 915
1200 IF LINPNT<2 THEN 970
1210 LINPNT=LINPNT-1:GOTO 915
1300 IF LINPNT=LINCNT THEN 970
1310 LINPNT=LINPNT+1:GOTO 915
1400 TRAP 660
1405 A$(23)="NEW VALUE=        "
1406 GOSUB 24000
1410 GET #1,M
1411 X=0:REM START WITH NO-HUNDREDS
1412 IF M<48 OR M>57 THEN 660:REM TRAP
     OUT ON BAD ENTRY
1414 POKE DLST+33,M-32
1416 X=M-48
1500 GET #1,M
1510 IF M=155 THEN POKE DV,X:GOTO 915
1520 IF M<48 OR M>57 THEN 660:REM TRAP
1530 Y=10*X+M-48
1540 POKE DLST+34,M-32
1550 GET #1,M
1560 IF M=155 THEN POKE DV,Y:GOTO 915
1570 IF M<48 OR M>57 THEN 660
1575 POKE DLST+35,M-32
1580 Z=10*Y+M-48
1590 IF Z>255 THEN 660
1600 POKE DV,Z:GOTO 915
3000 GRAPHICS 0:PRINT "DONE"
23000 FOR N=1 TO 192
```

```
23010 CUR(N)=0:NEXT N
23020 RETURN
24000 FOR N=1 TO LEN(A$)
24010 LTR=ASC(A$(N,N))
24020 IF LTR>97 AND LTR<123 THEN
      XX=LTR-0:GOTO 24050
24030 IF LTR>27 AND LTR<32 THEN
      XX=LTR+192:GOTO 24050
24040 XX=LTR-32
24050 POKE -1+N+DLST,XX
24060 NEXT N
24070 RETURN
25000 REM ENTRY VARIABLES MODE=MODE
25010 REM NORMALWIDE=TRUE IF VALUES
      FROM MEMPERLINE ARE TO BE USED,
      ELSE ARRAY SIZE IS IN
      MEMPERLINE(1)
25020 REM HSCROLL VSCROLL FALSE IF NO
      HS OR VS IS SPECIFIED
26000 IF MODE<2 OR MODE>15 THEN
      OK=FALSE:REASON=BADMODE:RETURN
26010 DINSTRUCT=MODE+64+HSCROLL*16+
      VSCROLL*32
26020 CTEMP=COUNT+LINESPERMODE(MODE)
26030 IF CTEMP>192 THEN OK=FALSE:
      REASON=TOOMANY:RETURN
26040 IF NORMALWIDE=FALSE THEN 26060
26050 DSTEMP=DS+MEMPERLINE(MODE):GOTO
      26070
26060 DSTEMP=DS+MEMPERLINE(1)
26070 IF DSTEMP>(XMEM*256) THEN
      OK=FALSE:REASON=OUTOFMEM:RETURN
26075 GOSUB 28000
26080 POKE DLPNT,DINSTRUCT
26081 REM PUT AWAY THE INSTRUCTION
26090 DHIGH=INT(DS/256):DLO=DS-
      DHIGH*256
27000 POKE DLPNT+1,DLO
```

```
27005 POKE DLPNT+2,DHIGH
27010 DLPNT=DLPNT+3
27020 COUNT=CTEMP:DS=DSTEMP:OK=
      TRUE:RETURN
28000 M=DFAULT(MODE)
28010 FOR N=DS TO DSTEMP-1
28020 POKE N,M
28030 NEXT N
28040 RETURN
30000 DATA 40,40,40,40,20,20,40,80,80
30010 DATA 160,160,160,160,320
30020 DATA 8,10,8,16,8,16,8,4,4,
      2,1,2,1,1
30030 DATA 45,45,69,69,85,85,85,85,85,
      85,85,85,85,85
```

Here is an explanation of the Edit mode. When you have defined one or more lines on the screen, each line will have some kind of data on it, just so that you can see the line there (if the data were zeros, the line would have been blank). The Edit mode allows you to move the cursor around on the screen through the data you defined, and change it to a new value if you wish.

Line 660 shows A$ as: Add Delete Edit eXit   . The A, D, E, and X should be entered as reverse video. When you enter this program line, press the **ATARI** key once just before you press each of the letters A, D, E, or X, then press it again before you continue to enter the rest of the word. This way only the selected letter will be in reverse video.

Because these characters are the only ones appearing in reverse, it can graphically show the user exactly which are his possible choices. The program is built to ignore any keystrokes except those shown in this way.

Line 661 says A$(36) = STR$(192 – COUNT). This means that this line should include the string equivalent of the number of lines which are still left to be defined. The user, in this case, can define up to 192 lines on the screen. As each mode is defined, it takes up one or more of the lines which are left. This adds to COUNT and subtracts from the number left (192 – COUNT). If

you try to define a screen with more than 192 lines, the program will end with an error.

Lines 662 and 663 handle any case when the count of leftover lines has fewer digits than the count just presented. It prevents a transition from, for example, 102 to 94 from being presented as "942." This is needed because the STR$ function begins with the leftmost character and fills in only as many as needed to present the whole number.

Line 670 calls subroutine 24000, which writes the data into the status line area of the memory. No matter where the status line appears on the screen, the same data will be used because that is what you have specified in the display list.

Line 710 reads the keyboard. It accepts a single character directly from the keyboard, rather than wait for the user to press **RETURN**. This was arranged by line 300, which OPENed IOCB#1 for keyboard reading.

Lines 715 through 730 give the various choices for routines to perform for various keyboard values. Notice that, if none of the possible choices are chosen, line 735 returns control to line 710; that is, wait for a key again and check again if it is one of those which are acceptable.

Line 750 is almost a duplicate of line 660. In fact, to enter it, you may type

```
LIST 660
```

then edit the line number into 750, and change the rest of the line as described here.

In line 750, the A, D, E, and X are *normal* video (not reversed), and the word "MODE =" is *reversed* video. In this way, when this line is printed into the status line area, the user is presented with a very specific, single choice of what kinds of keys will work. No keys other than the number keys, with values from 2 through 15, will be accepted.

If you are entering a mode from 10 through 15, then the first digit you enter will be a 1. That 1 is added to the status line and line 759 waits for the second key entry to define the mode.

Line 770 calls the routine which writes the default data into memory so that you will be able to see the new line you have defined.

The line-writing routine has several error codes which it can output. Such errors are:

1. Too many mode lines being used (more than 192).
2. Not enough memory to hold the display data. (Line 20 only reserves 4096 spaces, and a typical GRAPHICS 8 screen needs twice as many memory locations to produce the full screen. Additional data about this is contained in the suggestions section of this chapter.)

Line 771 adds one to the line counter. This assures that the editor will be able to get at all of the lines which have been defined so far.

Line 800 starts the BASIC program part for the Delete function. If the line counter is zero, then no lines can be deleted, so the command is ignored.

Line 801 moves the display list position pointer up one position. It subtracts 3 because each instruction in the display list takes up three spaces (command, low address, high address).

Lines 802 and 803 calculate, from the command information, how many lines are going to be reclaimed (made available again) when this last line is deleted. This data is being taken directly from the display list which is being constructed.

Lines 810 through 818 replace the mode definition and its address with an instruction to jump to show the status line. This eliminates the most recently defined line from the display list and moves the status line up one position on the screen.

Even though the video lines have been reclaimed in count, there is still one more task associated with the Delete function. That is to reclaim the memory which has been used for this most recently defined mode. Line 820 performs this function. The next line which will be added will therefore reuse the data memory area that has been vacated by the previously deleted mode.

Line 900 begins the Edit function. As a start, it will refuse to enter the Edit mode if there is nothing defined on the screen for it to edit.

Line 905 defines the new status line contents, showing M (for Mode number) for the L (for Line) number on which the edit cursor is sitting. C (for Cursor position) shows the horizontal count position within the line it is on, and V (for Value) shows what

the value of the data at that location actually is. On the screen, a graphic representation of the actual data appears in each of the defined screen positions.

Lines 915 through 957 calculate the values which are shown in the status line, then turn them into string constants so that a string handling subroutine (24000) can put them into the right position to appear in the status line.

Lines 970 through 999 get a user input without a **RETURN** keypress, then direct the program to the section which processes the key selected. Possible inputs are right cursor, left cursor, up cursor, down cursor, or N (for New value).

Each of the cursor moves used in lines 1000 through 1310 check if the cursor is already at the limits of its possible move. If it is possible to move the cursor in the desired direction, it will be done. Then the program cycles back through line 915 to report what happened on the status line.

If N is pressed, lines 1400 through 1600 change the mode line to show only **NEW VALUE =** in reversed video, and accepts a positive number from 0 to 255. The character at the current cursor position receives this new data value. It will show on the screen, as well as being reported in the status line.

SUMMARY OF THE PROGRAM

1. The purpose of the program is to show graphically how a display list is produced, and provides a tool which can be used to build a display list and manipulate its data.
2. It produces a specialized form of display list in which each line specifies the starting point of the memory where its data may be found.
3. The program includes the ability to enable the horizontal and vertical scrolling capability of the display processor; however, the status line prompts do not currently provide a selection for this mode. In addition, both the default data and the size of the memory per line is fixed in the DATA statements. A full display list builder would allow the size of the memory per line to be individually defined, to allow for the use of special ef-

fects. This was, however, beyond the scope of this book, and is left as a challenge to the user.
4. Once the user understands display lists, he or she may decide to modify this program to make it more useful or, perhaps, faster, or add commands to save or load a display list to or from the disk.

HOW TO USE THE PROGRAM

Type it in and SAVE it on disk or tape. Be sure you have entered the program correctly. This is especially necessary with any program which does POKEs directly to memory.

Type RUN. The screen will go blank and the program will type "WORKING . . ." at the top of the screen. At this time, it will be initializing some of its workspaces and reserving memory space for you to work with.

NOTE: If you decide, at any point, to stop this program and start it again, you MUST press **SYSTEM RESET** before you try to restart. This program changes location 106 to a number 16 lower than it was previously. Unless **SYSTEM RESET** is pressed, each time you **BREAK**, then RUN again, the contents of location 106 will keep getting lower and lower. You will run out of memory for your program. **SYSTEM RESET** restores this location to its original value for a new RUN.

When the status line comes up offering you Add, Delete, Edit, eXit—press A (Add).

When it asks **MODE =**—press 7. You will see the status line move down—a line of GRAPHICS 2 characters (5s) will appear on this line you have just defined. The status line returns to normal, but the number on the right side says 176. This indicates that you can define 176 more screen lines. The data shown in Table 7-1 shows you exactly how many video lines each of the modes will use, and how much of the data area each will use also.

This program does not account for data area on the status line. You have 3496 data positions available. Line 20 of the program, in combination with line 550, defines this total (each number value in location 106 defines 256 possible positions, so 16 * 256

= 4096, 4096 − 600 = 3496). If line 20 specified XMEM − 32, you would have 3496 + 4096 positions available, or 7592. This would be just about enough for a whole screen of GRAPHICS MODE 8 + 16.

Now Add a mode line 2. The sequence is:

A
2

Then the status line moves down again, this time putting in a whole line of capital Ms on a normal GRAPHICS MODE 0 line of 40 characters. This demonstrates that the mode lines which the display processor uses are actually different numbers than are normally referenced by BASIC. Table 7-1 indicates how the display processor modes relate to the GRAPHICS modes.

Now try to Delete a line (press D). The status line pops up one position and in a couple of seconds the count of available video lines is adjusted to reflect how many mode lines became available when it was deleted.

Add some more lines. Note that modes 8–15 take up very little space on the vertical screen part. *Be patient* when the program is adding these modes . . . there is more data to write, so it takes longer before the status line is back to its first state.

Now that there is some data there to edit, go into Edit mode (press E). If you still have mode 7 as the top line (the large 5s), you will see the status line read:

M = 7   L = 1   C = 1   V = 85   ←──→ ↑ ↓   **N**ewV.   **Q**uit

where, M = 7 says this is mode 7 on L = 1 (line 1) at C = 1 (cursor position points to leftmost character on this line), and V = 85 shows its current value.

Push the *right* cursor arrow key (no control key, just that cursor-labeled key directly). Each push of that key moves the C value one higher.

Push the *down* cursor key. Notice that this *increases* the L value by 1, but *changes* the C value to 1, from whatever value you had it before this keypress. You see, each horizontal line has its own edit cursor position. If you press the *up* cursor key, you will see the C value *jump back* to where it was to start with.

Push the N (New value) key. The status line changes to ask for the new value before going on. When you enter a value from 0 to 255, inclusive, if the value is 99 or less, press the **RETURN** key. Otherwise, just enter three digits (leading zeros such as 042, are all right to use). The graphics character at the position you have selected immediately changes to match the new value you have entered for it.

Try to change other characters on the screen the same way. Again, remember to be *patient*. This program is, after all, written in BASIC and has not been optimized.

When you are finished with this experimenting, press Q (Quit). It takes you back to the original status line. Press X (eXit).

Table 7-1. Display List Modes

Display Process Mode	Basic Graphics Mode	Kind of Display	Video Lines Used	Memory Per Modeline	No. of Colors Possible
2	0	characters	8	40	2.5
3	—	"	10	40	2.5
4	12	"	8	40	5
5	13	"	16	40	5
6	1	"	8	20	5
7	2	"	16	20	5
8	3	graphics	8	10	4
9	4	graphics	4	10	2
10	5	graphics	4	20	4
11	6	graphics	2	20	2
12	14	graphics	1	20	2
13	7	graphics	2	40	4
14	15	graphics	1	40	4
15	8	graphics	1	40	2.5

NOTES:

1. GRAPHICS MODES 12, 13, 14, and 15 (corresponding to display processor modes 4, 5, 12, and 14) are *not* present in the ATARI 400 and 800 computers. However, they *are* present in all of the XL series.

2. The colors which you may display on the screen are shown in your *ATARI BASIC Reference Manual*. These refer to the possible selection of the colors from COLOR REGISTERS 0 through 4.

3. When you see a designation "2.5" in the table, it means that slightly more than two different colors can be made to appear onscreen within a mode line such as this. In particular, the background color will appear as a border at the left and right edge of this display line. The color of the basic line will be selected from COLOR REGISTER 2, and the brightness of the characters or dots within this mode line will still have the color from COLOR REGISTER 2, but will have the brightness value from COLOR REGISTER 1. Thus, BACK-GROUND (1) plus COLOR2 (1) plus BRIGHTNESS1/COLOR2 (0.5) makes the possible color selections on this line "2.5."

Table 7-2. Display List Instructions

Instruction Value	Kind of Action Taken by Display Processor
——Special Instruction Modes——	
0 0 + 16n 1 65	DISPLAY 1 blank line. DISPLAY n + 1 blank lines. (Maximum instruction value 112, means display 8 blank lines.) JUMP to a new location, no other action (just like a BASIC GOTO). JUMP to a new location and wait for vertical blank (wait for the start of display of a new screen, happens once each 1/60 of a second).
——Display Instruction Modes——	
2–15	DISPLAY a new line using this display processor mode, assume that the data for this mode is stored in memory immediately following the data for the mode which was previously defined.

MODIFIERS FOR INSTRUCTIONS 2–15:

1. Add 64 if the memory-scan counter is to be loaded (define, in the instruction, where the data for this mode line is to start). Means that the instruction takes up 3 locations in the display list instead of just 1. This is the method used in the example program.

2. Add 32 to the instruction to enable vertical scrolling. POKEing a number from 0 to 15 into location 54277 will cause a vertical movement on any line which has this vertical scrolling enabled.
3. Add 16 to the instruction to enable horizontal scrolling of a line. POKEing a number from 0 to 15 into location 54276 causes a line with horizontal scrolling enabled to move left or right by a small amount.
4. Add 128 to the instruction to enable the "Display List Interrupt." This causes the display processor to interrupt the main processor to do something special, such as change the colors or perhaps change the movable object positions. This topic is more advanced and requires the use of machine language. Again see the list of suggested reading for more information on display lists.

THINGS TO WATCH OUT FOR

1. The display list cannot cross a 1024 location boundary in the memory. You can test for this condition by:

```
X = GPOS - ( INT (GPOS/1024) ) *
1024 - 3
```

If the value X reaches 0 as you are defining your display list, it means that the next place you wish to put a display list instruction will go across a 1024-byte boundary. The " − 3" is there to assure that you will have room in the display list to insert a JUMP instruction into the list, in place of a regular display instruction. Table 7-2 shows that a JUMP instruction is a "1," and is followed by the address to which the jump should take place—low value, then high value of the address (see the sample program, lines 445, 450). Once you have performed the jump, it reloads a memory counter inside of the display processor, and makes it ready to continue reading the display list at the new location.
2. The display data cannot cross a 4096-byte boundary. You can check this by:

```
X = DS - ( INT (DS/4096) ) * 4096 -
MEMPERLINE(MODE)
```

If X is less than 0, it means that the data will not fit into the memory space you might expect that it will be in. You must adjust DS this way:

```
DS = DS - X
```

Since X is a minus value, subtracting a minus value actually *adds* something to DS. Now you may proceed normally, since this correctly adjusts DS for the memory boundary crossing. (This is not in the demo program.)

REVIEW OF CHAPTER 7

This chapter has provided a tool, written in BASIC, which allows you to produce and experiment with display lists. A display list is a set of instructions which directs the actions of a separate display processor inside of your ATARI computer. You tell it HOW to display each video line or group of video lines on the screen, and WHERE to find the data. After writing all of the instructions, the list ends with a JUMP to the top of the list again. This tells the display processor to get ready for the next time the screen is to be produced.

If you define a standard sized screen, you may make the display list shorter by writing the list as:

DISPLAY 8 BLANK LINES
DISPLAY 8 BLANK LINES
DISPLAY 8 BLANK LINES
START A MODE X DISPLAY AT MEMORY LOCATION M
    DO ANOTHER MODE X (don't add 64 to the mode #)
    DO ANOTHER MODE X . . . , etc.,

for as many of those mode lines you may want.

    JUMP AND WAIT VERTICAL BLANK (instruction type 65)

specifying low and high value part of address where the display list starts.

APPENDIX

Suggestions for Further Reading

Here are listed some of the reference materials which have been useful during the preparation of the ATARI BASIC tutorials. You may wish to obtain them for your own use.

Mapping the ATARI. Ian Chadwick, Compute! Books, 1983.
This is a very comprehensive listing of the memory locations which have an effect on the way the ATARI 400 and 800 computers work. It also includes brief, but effective, sample programs which allow you to exercise certain functions. It can keep the curious function-browser busy for many hours.

DE RE ATARI. Chris Crawford, ATARI, Inc., 1982.
This book, published by ATARI, contains additional information about display lists, as well as an insight into some of the internal workings of ATARI BASIC. It also includes a quick reference card which has proved very useful in the writing of test programs for the 400 and 800.

The ATARI BASIC Source Book. Bill Wilkinson, Compute! Books, 1983.
This was written by the programmer of ATARI BASIC, and contains the complete source listing for the program itself. It is written

in machine code, but contains a full explanation showing how all of the individual routines work. Those of you who really want to get inside of ATARI BASIC might find this very useful.

ATARI Technical Reference Notes. ATARI, Inc., 1983.
    This is the definitive reference to the operating inner function. It may not be as "friendly" as the other references, but it contains information which is difficult to find from any other source.

Index

171

WHAT DO YOU DO AFTER YOU PLUG IT IN?

Complete tutorial covering use of microcomputer hardware, software, languages, operating systems, data communications, and more, followed by a second tutorial on workable solutions to the practical problems you'll meet while using them. Also has good advice on choosing a system and the software to run it.
William Barden, Jr.
200 pages, 5½ × 8½, softbound. ISBN 0-672-22008-3. ℠1983.
No. 22008. . . . . $10.95

USER'S GUIDE TO MICROCOMPUTER BUZZWORDS

A handy quick-reference for those people who don't care what happens inside a microcomputer yet who must know enough to be able to communicate with others who do. Provides an understanding of the basic terminology you need to become "computer literate." Contains many illustrations.
David H. Dasenbrock.
110 pages, 5½ × 8½, softbound. ISBN 0-672-22049-0. ℠1983.
No. 22049. . . . . $9.95

HOW TO MAINTAIN AND SERVICE YOUR SMALL COMPUTER

Shows you easy maintenance and operating procedures to sharply reduce possible problems with any small personal computer. In case of failure, it shows you how to diagnose what's wrong, identify the faulty part, and make many simple, money-saving repairs yourself. A basic knowledge of electronics is needed for most repairs.
Stephenson and Cahill.
224 pages, 8½ × 11, softbound. ISBN 0-672-22016-4. ℠1983.
No. 22016. . . . . $17.95

MEGABUCKS FROM YOUR MICROCOMPUTER

Shows you how to make money using your microcomputer for creative writing, reviewing, and programming. You'll learn about some dangers, get some tips on choosing the right microcomputer, and more.
Tim Knight.
80 pages, 8½ × 11, softbound. ISBN 0-672-22083-0. ℠1983.
No. 22083. . . . . $3.95

HOWARD W. SAMS CRASH COURSE IN MICROCOMPUTERS (2nd Edition)

An outstanding single-book, self-study course for those who need to know a lot in a hurry about microcomputers and programming. New chapters cover 8- and 16-bit microcomputing and BASIC programming, and an expanded applications chapter covers new software. Helps you in computer classes, too. No previous computer knowledge needed.
Louis E. Frenzel, Jr.
320 pages, 8½ × 11, comb-bound. ISBN 0-672-21985-9. ℠1983.
No. 21985. . . . . $21.95

ELECTRONICALLY SPEAKING: COMPUTER SPEECH GENERATION

Teaches you the basics of generating synthetic speech with an Apple® II, TRS-80®, or other popular microcomputer. Includes techniques, a synthesizer overview, advice on what you can do about possible problem areas, and a history of synthetic speech research since the 1800s.
John P. Cater
232 pages, 5½ × 8½, softbound. ISBN 0-672-21947-6. ℠1982.
No. 21947. . . . . $14.95

ELECTRONICALLY HEARING: COMPUTER SPEECH RECOGNITION

Brings you up to date on voice command over computers and makes it possible for you to construct a voice recognition system of your own from what you learn here. Clearly and understandably covers the practical aspects of computer speech analysis and recognition for beginners in the field, including necessary math and speech concepts. Also concentrates on software and hardware sysems and reviews what's available commercially.
John P. Cater.
272 pages, 5½ × 8½, softbound. ISBN 0-672-22173-X. ℠1984.
No. 22173. . . . . $13.95

USING COMPUTER INFORMATION SERVICES

Shows you how to use your microcomputer to communicate with the national computer networks and their wide range of services. Clearly explains what's available, how you can retrieve it, how to use your computer as a powerful communications tool, and more.
Sturtz and Williams.
240 pages, 5½ × 8½, softbound. ISBN 0-672-21997-2. ℠1983.
No. 21997. . . . . $12.95

These and other Sams Books and Software products are available from better retailers worldwide, or directly from Sams. Call 800-428-SAMS or 317-298-5566 to order, or to get the name of a Sams retailer near you. Ask for your free Sams Books and Software Catalog!

Prices good in USA only. Prices and page counts subject to change without notice.

Apple is a registered trademark of Apple Computer, Inc. ATARI is a registered trademark of ATARI, Inc. TRS-80 is a registered trademark of Radio Shack, a Tandy Corporation.

SAMS

Advanced ATARI® BASIC Tutorial

- Is written specifically for owners and users af ATARI computer systems
- Guides the reader step by step through advanced programming techniques via practical examples
- Picks up where **ATARI® BASIC Tutorial** leaves off, fallowing a progressive format—each chapter builds on knowledge gained in previous chapters
- Concentrates on the use of the ATARI Disk Operating System (DOS) and the commands necessary for efficient advanced programming techniques
- Helps users develop a working knowledge af the elements that make up typical data base management programs, as well as haw to modify and apply these routines ta the user's own programming needs
- Contains numeraus examples of debugged, self-documenting programs, including a variety of sart techniques, how to use arrays, and graphics applications of the ATARI camputer systems